09744790 .090302

JC02 Rec'd PCT/PTO 30 JAN 2001

FORM PTO-1390 (Modified) (REV 5-93) U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE		ATTORNEY'S DOCKET NUMBER
TRANSMITTAL LETTER TO THE UNITED STATES DESIGNATED/ELECTED OFFICE (DO/EO/US) CONCERNING A FILING UNDER 35 U.S.C. 371		042159/0117
		U.S. APPLICATION NO. (If known, see 37 C.F.R. 1.5) 09/744790
INTERNATIONAL APPLICATION NO. **PCT/US99/17369**	INTERNATIONAL FILING DATE **30 July 1999**	PRIORITY DATE CLAIMED **30 July 1998**

TITLE OF INVENTION
METHOD AND APPARATUS FOR DESIGN FORWARD ERROR CORRECTION TECHNIQUES IN DATA TRANSMISSION OVER COMMUNICATIONS SYSTEM

APPLICANT(S) FOR DO/EO/US
Juan Alberto TORRES, Victor DEMJANENKO and Frederic HIRZEL

Applicant herewith submits to the United States Designated/Elected Office (DO/EO/US) the following items and other information:

1. ☒ This is a FIRST submission of items concerning a filing under 35 U.S.C. 371.
2. ☐ This is a SECOND or SUBSEQUENT submission of items concerning a filing under 35 U.S.C. 371.
3. ☒ This express request to begin national examination procedures (35 U.S.C. 371(f)) at any time rather than delay examination until the expiration of the applicable time limit set in 35 U.S.C. 371(b) and PCT Articles 22 and 39(1).
4. ☒ A proper Demand for International Preliminary Examination was made by the 19th month from the earliest claimed priority date.
5. ☒ A copy of the International Application as filed (35 U.S.C. 371(c)(2))
   - ☐ is transmitted herewith (required only if not transmitted by the International Bureau).
   - ☐ has been transmitted by the International Bureau.
   - ☒ is not required, as the application was filed in the United States Receiving Office (RO/US)
6. ☐ A translation of the International Application into English (35 U.S.C. 371(c)(2)).
7. ☒ Amendments to the claims of the International Application under PCT Article 19 (35 U.S.C. 371(c)(3))
   - ☐ are transmitted herewith (required only if not transmitted by the International Bureau).
   - ☐ have been transmitted by the International Bureau.
   - ☐ have not been made; however, the time limit for making such amendments has NOT expired.
   - ☒ have been made and filed in the United States Receiving Office.
8. ☐ A translation of the amendments to the claims under PCT Article 19 (35 U.S.C. 371(c)(3)).
9. ☐ An oath or declaration of the inventor(s) (35 U.S.C. 371(c)(4)).
10. ☐ A translation of the annexes to the International Preliminary Examination Report under PCT Article 36 (35 U.S.C. 371(c)(5)).

Items 11. to 16. below concern other document(s) or information included:

11. ☐ An Information Disclosure Statement under 37 CFR 1.97 and 1.98.
12. ☐ An assignment document for recording. A separate cover sheet in compliance with 37 CFR 3.28 and 3.31 is included.
13. ☐ A FIRST preliminary amendment.
    ☐ A SECOND or SUBSEQUENT preliminary amendment.
14. ☐ A substitute specification.
15. ☐ A change of power of attorney and/or address letter.
16. ☐ Other items or information: Copy of International Search Report (2 pages)

09744790 . 090302

JC02 Rec'd PCT/PTO 30 JAN 2001

09/744790

U.S. APPLICATION NO. (If known, see 37 CFR 1.5) Unknown	INTERNATIONAL APPLICATION NO. PCT/US99/17369	ATTORNEY'S DOCKET NUMBER 042159/0117

17. ☒ The following fees are submitted:	CALCULATIONS	PTO USE ONLY
Basic National Fee (37 CFR 1.492(a)(1)-(5): Search Report has been prepared by the EPO or JPO $0.00		
International preliminary examination fee paid to USPTO (37 CFR 1.482) $690.00		
No international preliminary examination fee paid to USPTO (37 CFR 1.482) but international search fee paid to USPTO (37 CFR 1.445(a)(2) $0.00		
Neither international preliminary examination fee (37 CFR 1.482) nor International search fee (37 CFR 1.445(a)(2)) paid to USPTO $0.00		
International preliminary examination fee paid to USPTO (37 CFR 1.482) and all claims satisfied provisions of PCT Article 33(2)-(4) $0.00		
ENTER APPROPRIATE BASIC FEE AMOUNT =	$690.00	
Surcharge of $130.00 for furnishing the oath or declaration later than 20 Months from the earliest claimed priority date (37 CFR 1.492(e))		

Claims	Number Filed		Included in Basic Fee		Extra Claims		Rate	
Total Claims	32	-	20	=	12	×	$18.00	$216.00
Independent Claims	7	-	3	=	4	×	$78.00	$312.00
Multiple dependent claim(s) (if applicable)							$260.00	$0.00
TOTAL OF ABOVE CALCULATIONS =								$1218.00
Reduction by ½ for filing by small entity, if applicable. Verified Small Entity statement must also be filed. (Note 37 CFR 1.9, 1.27, 1.28).								$609.00
SUBTOTAL =								$609.00
Processing fee of $130.00 for furnishing English translation later the 20 months from the earliest claimed priority date (37 CFR 1.492(f). +								
TOTAL NATIONAL FEE =								$0.00
Fee for recording the enclosed assignment (37 CFR 1.21(h)). The assignment must be accompanied by an appropriate cover sheet (37 CFR 3.28, 3.31). $40.00 per property +								
TOTAL FEES ENCLOSED =								$609.00
							Amount to be: refunded	$
							charged	$

a. ☐ A check in the amount of $_____.00 to cover the above fees is enclosed.

b. ☒ Please charge my Deposit Account No. **50-0872** in the amount of **$609.00** to the above fees. A duplicate copy of this sheet is enclosed.

c. ☒ The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to Deposit Account No. 50-0872. A duplicate copy of this sheet is enclosed.

NOTE: Where an appropriate time limit under 37 CFR 1.494 or 1.495 has not been met, a petition to revive (37 CFR 1.137(a) or (b)) must be filed and granted to restore the application to pending status.

SEND ALL CORRESPONDENCE TO:

Foley & Lardner
2029 Century Park East, Suite 3500
Los Angeles, California 90067-3021
Telephone: (310) 975-7963

SIGNATURE Ronald Coslick

NAME RONALD COSLICK

REGISTRATION NUMBER 36,489

FORWARD ERROR CORRECTING SYSTEM WITH ENCODERS CONFIGURED IN PARALLEL AND/OR SERIES

This non-provisional patent application claims the benefit under 35 U.S.C. Section 119(e) of United States Provisional Patent Application No. 60/094,629, filed on July 30,1998, and Provisional Patent Application No. 60/098,394, filed on August 30,1998, and Provisional Patent Application No. 60/133,390, filed on May 10, 1999, all of which are incorporated herein by reference.

Field of the Invention

The present invention relates to the use of forward error correction techniques in data transmission over wired and wireless systems using an optional Reed-Solomon encoder as an outer encoder and a multiple concatenated convolutional encoder (in serial or parallel configuration) as an inner encoder. A preferred embodiment of the invention pertains particularly to ADSL systems, as a representative species of wired-based systems.

Background of the Invention

The invention is based on use of a multiple concatenated convolutional encoder in serial or in parallel configuration. In the serial case it is called Serial Multiple Concatenated Convolutional Code (SMCCC), and in the parallel case is called Parallel Multiple Concatenated Convolutional Code (PMCCC). This gives an extra redundancy to the signal in a way that improves the performance of the codification (increasing the coding gain).

Theory of Trellis Coding

Modulation constellations of more than 2 points (such as Quadrature Amplitude Modulation (QAM), and Quaternary Phase Shift Keying (QPSK)) are used to increase the bit rate at the cost of smaller Euclidean distances (distance between adjacent points in a signal constellation). Coding techniques are used to decrease transmission errors, when transmitting over power-limited channels.

Trellis Coding combines coding and modulation to improve bit error rate performance. As in other forms of forward error correction, the basic idea behind Trellis Coding is to introduce controlled redundancy in order to reduce channel error rates. What sets Trellis Codes apart is that this technique introduces redundancy by doubling the number of signal points in the QAM constellation (partitioning). BPSK and QPSK signal constellations are show in Figures 1 and 2.

The actual (noisy) received signal will tend to be somewhere around the "correct" signal point. The receiver chooses the signal point closest to the noisy received signal. As more points are added to the signal constellation and the power is kept constant, the probability of error increases, because the Euclidean distance (distance between adjacent signal points) "d" is decreased and the receiver has a more difficult job making the correct decision. Thus it would make sense, that the Euclidean distance "d" dominates the probability of error expressions.

Since the power is the same for both constellations, the required energy is also the same. The signal points in BPSK are $d = 2$ apart and $d = 1.414$ apart for QPSK.

The following expressions are for probability of error (for BPSK):

$$P_e = \frac{1}{2}\, erfc\left[\frac{d\sqrt{E}}{2\sqrt{no}}\right]$$

Substituting for "d" in the above:

$$P_e = \frac{1}{2}\, erfc\left[\frac{\sqrt{E}}{\sqrt{no}}\right] \text{(BPSK)}$$

$$P_e = \frac{1}{2}\, erfc\left[\frac{\sqrt{E}}{2\sqrt{no}}\right] \text{(QPSK)}$$

where *erfc* is the complementary error function.

Trellis Coding expands on this concept to increase the Euclidean path distance for a more thorough derivation of the probability of error. With the exception of the constant factor in front of the complimentary error function, it should be noted that both error expressions depend on the signal spacing d and that the probability for QPSK errors is higher (not surprising since the signal spacing is smaller). Trellis coding enables us to recover from this increase in probability of error.

M-QAM and PSK normally use a signal set of $M=2k$ symbols in order to reduce the symbol rate by a factor of M. Examples of $M=4$ QAM and QPSK are show in Figures 3 and 4 respectively.

Doubling the number of signal points in order to support two state Trellis Coding, we get the signal constellations for two state Trellis Coding shown in Figures 5 and 6.

Thus, Trellis Coding uses $2*M$ possible symbols for the same factor-of-M reduction of bandwidth (and each signal is still transmitted during the same signaling period).

Trellis Coding provides controlled redundancy, by doubling the number of signal points. In addition, Trellis coding defines the way in which signal transitions are allowed to occur. (Signal transitions that do not follow this scheme will be detected as errors).

This is best explained using the Trellis Coded *8-PSK* example. The *8-PSK* signal constellation is show in Figure 7, where we can see the individual signal points.

Note that the signal point labels "0,1,2,3,4,5,6,7" do not correspond to the actual data being sent. They are only convenient ways to label the signal points and keep from cluttering up the graphics.

Without coding, the performance in 8-PSK depends on d_0 ($d_0=2 \sin(\pi/8)=0.765$), which corresponds to a higher bit error rate than QPSK ($d_1=1.414$). By using Trellis Coding, it is possible to *improve* the performance by restricting the way in which signals are allowed to transition.

First the states of the trellis are defined. Lets label one state as "0426", and the other state as "1537". Each digit refers to one of four permitted signal points in the state (state points), with each state by itself representing a QPSK constellation, with each state's constellation being offset by 45 degrees from the other.

Figure 8 describes a two-state trellis, 8-PSK system. If the system is in state "0426" only one of these four state points is used. If a "0" or "4" is transmitted, the system remains in the same state. If however, a "2" or "6" is transmitted the system switches to the "1537" state. Now, only one of these four state points is used. If a "3" or "7" is transmitted, the system remains in this state, otherwise if a "1" or a "5" is transmitted, it switches back to the "0426" state. Again, note that the system in each symbol represents two bits, so that when switching states, the "*QPSK* constellation is shifted by 45 degrees".

Assuming that all input signals are equally likely, all signal paths are traced out over time. Just as we had for non-Trellis coding, the received signal includes noise and will tend to be located somewhere around the state points. The receiver again has to make a decision based on which signal point is closest and a mistaken output state value will be chosen if the receiver made an incorrect decision.

In order to illustrate error events, lets assume that the transmitter is sending continuous "7" symbols. Figure 9 illustrates the possible error events. In this case "5" followed by "6" is received instead of the transmitted "7" - "7" sequence. The Euclidean mean-squared distance for this path is the sum of the squares of the distance of each interval (see Figure 7 for an illustration of the Euclidean distances and Figure 8 for the Trellis Diagram):

$$\sqrt{d^2(7,5) - d^2(7,6)} = \sqrt{d_1^2 + d_0^2} = \sqrt{2 - \left(2 \sin\left(\frac{\pi}{8}\right)\right)^2} = 1.608$$

where $d(7,5)$ and $d(7,6)$ are the Euclidean distances between these two the signals "7" and "5", and "7" and "6" respectively.

Figure 10 show when case "1" followed by "6" is received instead of the transmitted "7" - "7" sequence. The Euclidean distance (see Figure 7 for an illustration of the Euclidean distances) for this path:

$$\sqrt{d^2(7,1) + d^2(7,6)} = \sqrt{d_1^2 + d_0^2} = \sqrt{2 + \left(2 \sin\left(\frac{\pi}{8}\right)\right)^2} = 1.608$$

Figure 11 show when case "5" followed by "2" is received instead of the transmitted "7" - "7" sequence. The Euclidean distance (see Figure 7 for an illustration of the Euclidean distances and Figure 8 for the Trellis Diagram) for this path is:

$$\sqrt{d^2(7,5) + d^2(7,2)} = \sqrt{d_1^2 + d_3^2} = \sqrt{2 + \left(2 \sin\left(\frac{3\pi}{8}\right)\right)^2} = 2.33$$

Figure 12 show when case "1" followed by "2" is received instead of the transmitted "7" - "7" sequence. The Euclidean distance (see Figure 7 for an illustration of the Euclidean distances and Figure 8 for the Trellis Diagram) for this path is:

$$\sqrt{d^2(7,1) + d^2(7,2)} = \sqrt{d_1^2 + d_3^2} = \sqrt{2 + \left(2 \sin\left(\frac{3\pi}{8}\right)\right)^2} = 2.33$$

The only remaining error event is the single interval "3" instead of "7" error event, which has a Euclidean distance of 2 (see Figure 7).

Because of their large Euclidean distance (2.33), the "5" - "2" and "1" - "2" error events are least likely. The "1" - "6" and "5" - "6" error events are most likely because of their low Euclidean distance (1.608).

The minimum Euclidean distances for a trellis is the minimum free Euclidean distance "d_E" (similar to the minimum free distance in convolutional coding). For the above example $d_E = 1.608$.

Since d_E is a measure of the closest spacing between adjacent state points (and therefore also more likely to cause errors), it dictates the lower bound for probability of error for the entire Trellis in the following way:

$$P_e \geq a(d_E)\frac{1}{2} erfc\left[d_E \frac{\sqrt{E}}{2\sqrt{n_0}}\right]$$

where, $a(d_E)$is the number of error paths at distance d_E. In the 2 states trellis example, there are 2 error paths at a distance of d_E. Therefore the probability of error is:

$$P_e \geq erfc\left[\frac{1.608}{2}\sqrt{\frac{E}{n_0}}\right]$$

We found the probability of error for regular (non Trellis coded) QPSK to be:

$$P_e = erfc\left[\frac{1.404}{2}\sqrt{\frac{E}{n_0}}\right]$$

The improvement of two state Trellis Coding over QPSK is therefore 1.608/1.414 or 1.1 dB.

This is a low coding gain for the amount of overhead required to handle Trellis coding. One might ask, is there a way to increase the coding gain obtainable with Trellis Coding? There certainly is. First, it is possible to increase the number of trellis states above 2, such as the four-state trellis shown in Figure 13. Note that the permitted state transitions are only drawn for column "I". The same state transitions are gain permitted in column "II", "III", and so on.

Increasing the number of states, one also increases the Euclidean distances. For example, in the above four state trellis coding case, all error events have a Euclidean distance of more than 2 (single "4" error). This single " 4" error, is the only "lowest" error event with minimum Euclidean path distance d_E

Therefore, the lower bound for 4 state Trellis Coding is:

$$P_e \geq \frac{1}{2} erfc\left[\sqrt{\frac{E}{n_0}}\right]$$

Comparing this equation to the equation for regular QPSK, we have a coding gain of 2:1 or 3 dB.

Table 1 illustrates further coding gains that can be obtained by using even more states in the trellis:

Table 1. Trellis Coding Gain vs. Number of Trellis States

# of Trellis States	Coding Gain
4	3.54
8	4.01
16	4.44
32	5.13
64	5.33
128	5.33
256	5.51

Another way to improve coding gain in Trellis Coding is to go to more than 2 dimensions.

Summary of the Invention

The present invention comprises forward error correction techniques in data transmission over wired systems using an optional Reed Solomon encoder as an outer encoder and a multiple concatenated convolutional encoder (MCCC) (in serial or parallel configuration) as an inner encoder. With an "optional" Reed-Solomon outer encoder we mean that it could be present or not. We describe its application to ADSL DMT (Discrete Multi-Tone, multiple-carrier) based systems. The extension to CAP/QAM (single-carrier) based, other xDSL systems (HDSL, VDSL, HDSL2, etc.), other wired communication systems, wireless systems and satellite systems is straightforward.

ADSL modems are designed to operate between a Central Office CO (or a similar point of presence) and a customer premises CPE. As such they use existing telephone network wiring between the CO and the CPE. There are several modems in this class which function in generally similar manner. All of these modems transmit their signals usually above the voice band. As such, they are dependent on adequate frequency response above voice band.

With the technique that we propose in this invention, it is possible to reach longer loops or reduce the transmitter power for ADSL systems.

For wireless systems it is possible to reduce the power consumption, increase the coverage area and to extend the life of the portable systems.

For satellite systems it is possible to increase the G/T factor around 4 dB, to increase the life of the satellite, to increase the coverage area and to reduce the requirements of the terrestrial systems.

With the use of the Trellis Coded Modulation (TCM) it is possible to obtain coding gains between 3 and 6 dB (depending on the dimension of the trellis). Using the technique that we propose, the performance of this technique is within 1 dB from the Shannon limit, at a bit error probability of 10^{-7}.

MCCC achieves near-Shannon-limit error correction performance. We have done some simulations that show bit error probabilities as low as 10^{-5} at E_b/N_o=0.6 dB. PMCCC yield very large coding gains (around 10 or 11 dB). In the PMCCC case, after this value of 10^{-5}, the role of the interleaver is very critical and to avoid the floor-error it is necessary to make a good design of the interleaver. In our design the Reed-Solomon outer encoder help to take care of this floor-error lower than 10^{-7}.

In this invention, we present simulation results, and we compare the Reed-Solomon encoder (for R=8 and R=16) with the Trellis plus Reed-Solomon (T+R=8 and T+R=16) and the two PCCC plus Reed-Solomon (TC+R=8 and TC+R=16). In all this cases will not take into account the payload of the Reed-Solomon code, because will have the same effect in all coding techniques.

These results, we present, are for Gaussian noise. In many wire-line systems, broadband impulse noise is also a significant transmission impairment. Although we have not modeled impulse noise effects in this analysis, in DMT systems, impulse noise whose duration is short compared to the frame size appears to be rather Gaussian-like, since it passes through a DFT in the receiver. Furthermore, because the noise is broadband, the noise energy in the signal band is distributed among the various frequency bins. Thus the additional immunity against additive white Gaussian noise provided by the trellis code should be beneficial for impulse noise as well.

We present an encoder, decoders and some simulation results.

Brief Description of the Several Views of the Drawings

Figure 1 shows a BPSK signal constellation;

Figure 2 shows a QPSK signal constellation;

Figure 3 shows a QAM signal constellation with M=4;

Figure 4 shows a QPSK signal constellation;

Figure 5 shows a QAM signal constellation with M=8 (Used with two state Trellis);

Figure 6 shows a PSK signal constellation with M=8 (Used with two state Trellis);

Figure 7 shows a signal constellation with M=8 (Used with two state Trellis);

Figure 8 shows a two-state Trellis 8-PSK system;

Figure 9 shows an error Event "5" → "6" in a 2 states Trellis encoding;

Figure 10 shows an error Event "1" → "6" in a 2 states Trellis encoding;

Figure 11 shows an error Event "5" → "2" in a 2 states Trellis encoding;

Figure 12 shows an error Event "1" → "2" in a 2 states Trellis encoding;

Figure 13 shows a four-state Trellis, 8-PSK system;

Figure 14 shows a serial Concatenated (n,k,N) block code;

Figure 15 shows the action of a uniform interleaver of length 4 on sequences of weight 2;

Figure 16 shows a serially Concatenated (n,k,N) Convolutional code;

Figure 17 shows a code sequence in $A_{h,j}$;

Figure 18 shows an analytical bounds for SMCBC1 for N = 4, 40, 400 and 4000;

Figure 19 shows an analytical bounds for SMCBC2 for N = 5, 50, 500 and 5000;

Figure 20 shows an analytical bounds for SMCBC3 for N = 7, 70, 700 and 7000;

Figure 21 shows an analytical bounds for SMCCC1 for N = 200, 400, 600, 800, 1000 and 2000;

Figure 22 shows an analytical bounds for SMCCC2 for N=200, 400, 600, 800, 1000, 2000;

Figure 23 shows an analytical bounds for SMCCC3 for N=200, 400, 600, 800, 1000, 2000;

Figure 24 shows an analytical bounds for SMCCC4;

Figure 25 shows a PMCCC;

Figure 26 shows a transmission system structure;

Figure 27 shows notations in a transmission system structure;

Figure 28 shows a PMCCC of three convolutional codes;

Figure 29 shows a signal flow graph for extrinsic information;

Figure 30 shows an iterative decoder structure for three parallel concatenated codes;

Figure 31 shows an iterative decoder structure for two parallel concatenated codes;

Figure 32 shows a convergence of turbo coding: bit-error probability versus number of iterations for various E_b/N_0 using the SW2-BCJR algorithm;

Figure 33 shows a convergence of turbo coding: bit-error probability versus number of iterations for various E_b/N_0 using the SWAL2-BCJR algorithm;

Figure 34 shows a bit-error probability as a function of the bit signal-to-noise ratio using the SW2-BCJR and SWAL2-BCJR algorithms with five iterations;

Figure 35 shows a number of iterations to achieve several bit-error probabilities as a function of the bit signal-to-noise ratio using the SWAL2-BCJR algorithm;

Figure 36 shows a Number of iterations to achieve several bit-error probabilities as a function of the bit signal-to-noise ratio using the SW2-BCJR algorithm;

Figure 37 shows a basic structure for backward computation in the log-BCJR MAP algorithm;

Figure 38 shows a Trellis Termination;

Figure 39 shows an example where a block interleaver fails to "break" the input sequence;

Figure 40 shows the two PMCCC performance, $r = ¼$;

Figure 41 shows performance with short block sizes;

Figure 42 shows three-code performance;

Figure 43 shows a comparison of SMCBC and PMCBC with various interleaver lengths chosen so as to yield the same input decoding delay;

Figure 44 shows a comparison of SMCCC and PMCCC with four-state MCCs;

Figure 45 shows Block diagram of a parallel concatenated convolutional code (PMCCC) (a) a PMCCC rate =1/3 (b) iterative decoding of a PMCCC;

Figure 46 shows Block diagram of a serial concatenated convolutional code (SMCCC) (a) an SMCCC rate =1/3 (b) iterative decoding of an SMCCC;

Figure 47 shows a trellis encoder;

Figure 48 shows an edge of the trellis section;

Figure 49 shows the soft-input soft-output (SISO) model;

Figure 50 shows the convergence of PMCCC-decoding: bit error probability versus the number of iterations using the ASW-SISO algorithm;

Figure 51 shows the convergence of iterative decoding for a serial concatenated code: bit error rate probability versus number of iterations using the ASW-SISO algorithm;

Figure 52 shows a comparison of two rate 1/3 PMCCC and SMCCC. The curves refer to six and nine iterations of the decoding algorithm and to an equal input decoding delay of 16,384;

Figure 53 shows a block diagram for a modem transmitter in accordance with this invention, for the Central Office and for STM transport;

Figure 54 shows a block diagram for a modem transmitter in accordance with this invention, for the Central Office and for ATM transport;

Figure 55 shows a block diagram for a modem transmitter in accordance with this invention, for the Remote modem and for STM transport;

Figure 56 shows a block diagram for a modem transmitter in accordance with this invention, for the Remote modem and for ATM transport;

Figure 57 shows an ATU-C functional interfaces for STM transport at the V-C reference point;

Figure 58 shows an ATU-C functional interfaces to the ATM layer at the V-C reference point;

Figure 59 shows an ATM cell delineation state machine;

Figure 69 shows an Example implementation of the $\Delta^2 f$ measurement;
Figure 61 shows an ADSL superframe structure – ATU-C transmitter;
Figure 62 shows a fast synchronization byte ("fast byte") format – ATU-C transmitter;
Figure 63 shows an interleaved synchronization byte ("sync byte") format – ATU-C transmitter;
Figure 64 shows a fast data buffer – ATU-C transmitter;
Figure 65 shows an interleaved data buffer, ATU-C transmitter;
Figure 66 shows a scrambler;
Figure 67 shows a tone ordering and bit extraction example (without trellis coding);
Figure 68 shows a tone ordering and bit extraction example (with trellis coding);
Figure 69 shows a conversion of u to v and w;
Figure 70 shows a finite state machine for Wei's encoder;
Figure 71 shows a convolutional Encoder;
Figure 72 shows a trellis diagram;
Figure 73 shows a constellation labels for $b = 2$ and $b = 4$;
Figure 74 shows an expansion of point n into the next larger square constellation;
Figure 75 shows a constellation labels for $b = 3$;
Figure 76 shows a constellation labels for $b = 5$;
Figure 77 shows a MTPR test;
Figure 78 shows an ATU-R functional interfaces for STM transport at the T-R reference point;
Figure 79 shows an ATU-R functional interfaces to the ATM layer at the T-R reference point;
Figure 80 shows a fast data buffer – ATU-R transmitter;
Figure 81 shows an interleaved data buffer – ATU-R transmitter;
Figure 82 shows two parallel Concatenated convolutional Encoder;
Figure 83 shows a conversion of u to v and w in the PMCCC encoder;
Figure 84 shows a decoder for PMCCC;
Figure 85 shows the convergence of "constellation" interleaver for PMCCC;
Figure 86 shows an interleaver for PMCCC;
Figure 87 shows a Serial Convolutional Concatenated Encoder;
Figure 88 shows a decoder for SMCCC;
Figure 89 shows an interleaver for SMCCC;
Figure 90 shows the Convolutional Concatenated Encoder used for simulations;
Figure 91 shows simulations for PMCCC; and,
Figure 92 shows the Convolutional encoder uses for simulations.

Detailed Description of the Preferred Embodiment

We hereby incorporate by reference the following references:

1. Rauschmayer, Dennis J. "ADSL/VDSL Principles". Macmillan Technical Publishing, 1999.
2. ITU G.992.1 "ADSL Transceivers", ITU, 1999.
3. ITU I.432 "B-ISDN user-network interface-physical layer specification", ITU , 1993.
4. Benedetto, Divsalar, Montorsi and F. Pollara, "Serial Concatenation of Interleaved Codes: Performance Analysis, Design, and Iterative Decoding", The Telecommunications and Data Acquisition Progress Report 42-126, Jet Propulsion Laboratory, Pasadena, California, pp. 1-26, August 15, 1996.

5. Benedetto, Divsalar, Montorsi and F. Pollara, "A Soft-Output Maximum A Posteriori (MAP) Module to decode parallel and Serial Concatenated Codes", The Telecommunications and Data Acquisition Progress Report 42-127, Jet Propulsion Laboratory, Pasadena, California, pp. 1-20, November 15, 1996.

6. L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," IEEE Transactions on Information Theory, pp. 284-287, March 1974.

7. Divsalar and F. Pollara, "Turbo Codes for PCS Applications", Proceedings of ICC'95, Seattle, Washington, pp. 54-59, June 1995.

8. D. Divsalar and F. Pollara, "Multiple Turbo Codes", Proceedings of IEEE MILCOM95, San Diego, California, November 5-8, 1995.

9. D. Divsalar and F. Pollara, "Soft-Output Decoding Algorithms in iterative Decoding of Turbo Codes," The Telecommunications and Data Acquisition Progress Report 42-124, Jet Propulsion Laboratory, Pasadena, California, pp. 63-87, February 15, 1995.

1. Performance Analysis, design and iterative decoding of SMCCC and PMCCC.

1.1 Analytical Bounds to the Performance of Serially Multiple Concatenated Codes.

1.1.1. Serially Multiple Concatenated Block Codes (SMCBC).

The scheme of two serially concatenated block codes is shown in Figure 14. It is composed of two cascaded CCs, the outer *(N,k)* code C_o with rate $R^o_c = k/N$ and the inner *(n, N)* code C_i with rate $R^i_c = N/n$, linked by an interleaver of length *N*. The overall SMCBC is then an *(n, k)* code, and we will refer to it as the *(n, k, N)* code C_S, including also the interleaver length. In the following, we will derive an upper bound to the ML performance of the overall code C_S. We assume that the CCs are linear, so that the SMCBC also is linear and the uniform error property applies, i.e., the bit-error probability can be evaluated assuming that the all-zero codeword has been transmitted.

A crucial step in the analysis comprises of replacing the actual interleaver that performs a permutation of the *N* input bits with an abstract interleaver called "uniform interleaver". This abstract interleaver is defined as a probabilistic device that maps a given input word of weight *l* into all distinct permutations of it, with equal probability $p = 1/\binom{N}{l}$ (see Figure 15). The output-word of the outer code and the input word of the inner code share the same weight. Use of the uniform interleaver permits the computation of the "average" performance of SMCBCs, intended as the expectation of the performance of SMCBCs using the same MCCs, taken over the ensemble of all interleavers of a given length. It can be proof the meaningfulness of the average performance, in the sense that there will always be, for each value of the signal-to-noise ratio, at least one particular interleaver yielding performance better than or equal to those of the uniform interleaver.

Let us define the input-output weight enumerating function (IOWEF) of the SMCBC C_S as

$$A^{C_S}(W,H) = \sum_{w,h} A^{C_S}_{w,h} W^w H^h \qquad (1)$$

where $A^{C_S}_{w,h}$ is the number of codewords of the SMCBC with weight *h* associated with an input word of weight *w*. We also define the conditional weight enumerating function (CWEF) $A^{C_S}(w,H)$ of the SMCBC as the weight distribution of codewords of the SMCBC that have input word weight *w*. It is related to the IOWEF by

$$A^{C_S}(w,H) = \left(\frac{1}{w!} \frac{\delta^w A^{C_S}(W,H)}{\delta W^w} \right)_{W=0} \qquad (2)$$

With knowledge of the CWEF, an upper bound to the bit-error probability of the SMCBC can be obtained in the form

$$P_b(e) \le \left(\sum_{w=1}^{k} \frac{w}{k} A^{C_S}(w,H) \right)_{H = e^{-\left(\frac{R_c E_b}{N_o}\right)}} \qquad (3)$$

where $R_c = k/n$ is the rate of C_S, and E_b/N_o is the signal-to-noise ratio per bit.

The problem comprises in the evaluation of the CWEF of the SMCBC from the knowledge of the CWEFs of the outer and inner codes, which we call $A^{C_o}(w, L)$ and $A^{C_i}(l, H)$. To do this, we exploit the properties of the uniform interleaver, which transforms a codeword of weight l at the output of the outer encoder into all its distinct $\binom{N}{l}$ permutations. As a consequence, each codeword of the outer code C_o of weight l, through the action of the uniform interleaver, enters the inner encoder generating $\binom{N}{l}$ codewords of the inner code C_i. Thus, the number $A^{C_s}_{w,h}$ of codewords of the SMCBC of weight h associated with an input word of weight w is given by

$$A^{C_s}_{w,h} = \sum_{l=0}^{N} \frac{A^{C_o}_{w,l} \; x \; A^{C_i}_{l,h}}{\binom{N}{l}} \qquad (4)$$

From Equation (4), we derive the expressions of the IOWEF and CWEF of the SMCBC:

$$A^{C_s}(w, H) = \sum_{l=0}^{N} \frac{A^{C_o}_{w,l} \; x \; A^{C_i}(l, H)}{\binom{N}{l}} \qquad (5)$$

$$A^{C_s}(W, H) = \sum_{l=0}^{N} \frac{A^{C_o}(W, l) \; x \; A^{C_i}(l, H)}{\binom{N}{l}} \qquad (6)$$

where $A^{C_o}(W, l)$ is the conditional weight distribution of the input words that generate codewords of the outer code of weight l.

1.1.2. Serially Multiple Concatenated Convolutional Codes

The structure of a serially multiple concatenated convolutional code (SMCCC) is shown in Figure 16. It refers to the case of two convolutional CCs, with the outer code C_o with rate $R^o_c = k/p$, and the inner code C_i with rate $R^i_c = p/n$, joined by an interleaver of length N bits. In this way they generating an SMCCC C_S with rate $R_c = k/n$. Note that N must be an integer multiple of p. We assume, as before, that the convolutional CCs is linear, so that the SMCCC is linear as well, and the uniform error property applies. The exact analysis requires the use of a hypertrellis having as hyperstates pairs of states of outer and inner codes. The hyperstates $S_{i,j}$ and $S_{l,m}$ are joined by a hyperbranch that comprises of all pairs of paths with length N/p that join states s_i and s_l of the inner code and states s_j and s_m of the outer code, respectively. Each hyperbranch is thus an equivalent SMCBC labeled with an IOWEF that can be evaluated as explained in the previous subsection. From the hypertrellis, the upper bound to the bit-error probability can be obtained through the standard transfer function technique employed for convolutional codes.

1.2. Design of Serially Multiple Concatenated Codes

For practical applications, SMCCCs are to be preferred to SMCBCs. One reason is that maximum a posteriori algorithms are less complex for convolutional than for block codes: a second is that the interleaver gain can be greater for convolutional CCs, provided they are suitably designed. Hence, we deal mainly with the design of SMCCCs, extending our conclusions to SMCBCs when appropriate.

Consider the SMCCC depicted in Figure 16. Its performance can be approximated by that of an equivalent block code whose IOWEF labels the branch of the hypertrellis joining the zero states of the outer and inner codes. Denoting by $A^{C_S}(w, H)$ the CWEF of this equivalent block code, we can rewrite the upper bound, Equation (3), as (subscript m will denote minimum, and a subscript M will denote maximum)

$$P_b(e) \le \left(\sum_{w=w_m^o}^{N R_c^o} \frac{w}{N R_c^o} A^{C_s}(w,H) \right)_{H=e^{-\left(\frac{R_c E_b}{N_o}\right)}} = \sum_{h=h_m}^{\frac{N}{R_c^i}} \sum_{w=w_m^o}^{N R_c^o} \frac{w}{N R_c^o} A_{w,h}^{C_s} e^{-\frac{h R_c E_b}{N_o}} \quad (7)$$

where w^o_m is the minimum weight of an input sequence generating an error event of the outer code, and h_m is the minimum weight (since the input sequences of the inner code are not unconstrained independent identically distributed (i.i.d.) binary sequences but, instead, codewords of the outer code, h_m can be greater than the inner code free distance, d^i_f) of the codewords of C_S. By "error event of a convolutional code" we mean a sequence diverging from the zero state at time zero and remerging into the zero state at some discrete time $j > 0$. For constituent block codes, an error event is simply a codeword.

The coefficients $A^{C_s}_{w,h}$ of the equivalent block code can be obtained from Equation (4) once the quantities $A^{C_o}_{w,l}$ and $A^{C_i}_{l,h}$ of the CCs are known. To evaluate them, consider a rate $R=p/n$ convolutional code C with memory ν, and its equivalent $(N/R\ ,\ N - p\nu)$ block code whose codewords are all sequences of length N/R bits of the convolutional code starting from and ending at the zero state. By definition, the codewords of the equivalent block code are concatenations of error events of the convolutional codes. Let

$$A(l,H,j) = \sum_h A_{l,h,j} H^h \quad (11)$$

be the weight enumerating function of sequences of the convolutional code that concatenate j error events with total input weight l (see Figure 17), where $A_{l,h,j}$ is the number of sequences of weight h, input weight l, and number of concatenated error events j. For N much larger than the memory of the convolutional code, the coefficient $A^C_{l,h}$ of the equivalent block code can be approximated (this assumption permits neglecting the length of error events compared to N , which also assumes that the number of ways j input sequences producing j error events can be arranged in a register of length N is $\binom{N/p}{j}$. The ratio N/p derives from the fact that the code has rate p/n, and thus N bits corresponds to N/p input words or, equivalently, trellis steps) by

$$A^C_{w,l} \sim \sum_{j=l}^{n_M} \binom{N/p}{j} A_{l,h,j} \quad (12)$$

where n_M , the largest number of error events concatenated in a codeword of weight h and generated by a weight l input sequence, is a function of h and l that depends on the encoder. Let us return now to the block code equivalent to the SMCCC. Using the previous result of Equation (12) with $j = n^i$ for the inner code, and the analogous one, $j=n^o$, for the outer code (superscripts o and i will refer to quantities pertaining to outer and inner code, respectively),

$$A^{C_o}_{w,l} \sim \sum_{n^o=l}^{n^o_M} \binom{N/p}{n^o} A^o_{w,l,n^o} \quad (13)$$

and substituting them into Equation (4), we obtain the coefficient $A^{C_s}_{w,h}$ of the serially concatenated block code equivalent to the SMCCC in the form

$$A^{C_s}_{w,h} \sim \sum_{l=d^o_f}^{N} \sum_{n^o=l}^{n^o_M} \sum_{n^i=l}^{n^i_M} \frac{\binom{N/p}{n^o}\binom{N/p}{n^i}}{\binom{N}{l}} A^o_{w,l,n^o} A^i_{l,h,n^i} \quad (14)$$

where d^o_f is the free distance of the outer code. By free distance d_f we mean the minimum Hamming weight of error events for convolutional CCs and the minimum Hamming weight of codewords for block CCs. We are interested in large interleaver lengths and thus use for the binomial coefficient the asymptotic approximation

$$\binom{N}{n} \sim \frac{N^n}{n!}$$

Substitution of this approximation in Equation (14) yields

$$A^{CS}_{w,h} \sim \sum_{l=d^o_f}^{N} \sum_{n^o=1}^{n^o_M} \sum_{n^i=1}^{n^i_M} \frac{l!}{p^{n^o+n^i}\, n^o!\, n^i!}\, A^o_{w,l,n^o}\, A^i_{l,h,n^i} \quad (15)$$

Finally, substituting Equation (15) into Equation (10) gives the bit-error probability bound in the form

$$P_b(e) \lesssim \sum_{h=h_m}^{\frac{N}{R_c}} e^{\left(hR_c\frac{E_b}{N_o}\right)} \sum_{w=w^o_m}^{NR^o_c} \sum_{l=d^o_f}^{N} \sum_{n^o=1}^{n^o_M} \sum_{n^i=1}^{n^i_M} N^{n^o+n^i-l-1} \frac{l!}{p^{n^o+n^i}\, n^o!\, n^i!}\, \frac{w}{k} A^o_{w,l,n^o}\, A^i_{l,h,n^i} \quad (16)$$

Using Expression (16) as the starting point, we will obtain some important design considerations. The bound, Expression (16), to the bit-error probability is obtained by adding terms of the first summation with respect to the SMCCC weights h. The coefficients of the exponential in h depend, among other parameters, on N. For large N, and for a given h, the dominant coefficient of the exponential in h is the one for which the exponent of N is maximum. Define this maximum exponent as

$$\alpha(h) = \max_{w,l} \left\{ n^o + n^i - l - 1 \right\} \quad (17)$$

Evaluating $\alpha(h)$ in general is not possible without specifying the CCs. Thus, we will consider two important cases for which general expressions can be found.

1.2.1. The Exponent of N for the Minimum Weight

For large values of E_b/N_0, the performance of the SMCCC is dominated by the first term of the summation in h, corresponding to the minimum value $h = h_m$. Remembering that, by definition, n^i_M and n^o_M are the maximum number of concatenated error events in codewords of the inner and outer code of weights h_m and l, respectively, the following inequalities hold true:

$$n^i_M \le \left\lfloor \frac{h_m}{d^i_f} \right\rfloor \quad (18)$$

$$n^o_M \le \left\lfloor \frac{l}{d^o_f} \right\rfloor \quad (19)$$

and

$$\alpha(h_m) \le \max_{l} \left\{ \left\lfloor \frac{h_m}{d^i_f} \right\rfloor + \left\lfloor \frac{l}{d^o_f} \right\rfloor - l - 1 \right\} = \left\lfloor \frac{h_m}{d^i_f} \right\rfloor + \left\lfloor \frac{l_m(h_m)}{d^o_f} \right\rfloor - l_m(h_m) - 1 \quad (20)$$

where $l_m(h_m)$ is the minimum weight l of codewords of the outer code yielding a codeword of weight h_m of the inner code, and $\lfloor x \rfloor$ means "integer part of x"(floor value) In most cases, $l_m(h_m) < 2d^o_f$ and $h_m < 2d^i_f$, so that $n^i_M = n^o_M = 1$ and Equation (20) becomes

$$\alpha(h_m) = 1 - l_m(h_m) \le 1 - d^o_f \quad (21)$$

The result, Equation (21), shows that the exponent of N corresponding to the minimum weight of SMCCC codewords is always negative for $2 \leq d_f^o$, thus yielding an interleaver gain at high E_b/N_0. Substitution of the exponent $\alpha(h_m)$ into Expression (16) truncated to the first term of the summation in h yields

$$\lim_{\frac{E_b}{N_o}\to\infty} P_b(e) \leq B_m \, N^{1-d_f^o} \, e^{-\left(\frac{h_m R_c E_b}{N_o}\right)} \qquad (22)$$

where the constant B_m is

$$B_m = \frac{A_{l_m(h_m),h_m,l}\,[l_m(h_m)]!}{kp} \sum_{w\in W_m} w \, A^o_{w,l_m(h_m),l}$$

and W_m is the set of input weights w that generates codewords of the outer code with weight $l_m(h_m)$. Expression (22) suggests the following conclusions:

1. For the values of E_b/N_0 and N where the SMCCC performance is dominated by its free distance $d_f^{C^s} = h_m$, increasing the interleaver length yields a gain in performance.
2. To increase the interleaver gain, one should choose an outer code with a large d_f^o.
3. To improve the performance with E_b/N_0, one should choose an inner and outer code combination such that h_m is large.

These conclusions do not depend on the structure of the CCs, and thus they apply for both recursive and nonrecursive encoders.

However, for a given E_b/N_0, there seems to be a minimum value of N that forces the bound to diverge. In other words, there seem to be coefficients of the exponents in h, for $h > h_m$, that increase with N. To investigate this phenomenon, we will evaluate the largest exponent of N, defined as

$$\alpha_M = \max\{\alpha(h)\} = \max\{n^o + n^i - l - l\} \qquad (23)$$

This exponent will permit one to find the dominant contribution to the bit-error probability for $N\to\infty$

1.2.2. The Maximum Exponent of N

We need to treat the cases of nonrecursive and recursive inner encoders separately. As we will see, nonrecursive encoders and block encoders show the same behavior.

1.2.2.1. Block and Nonrecursive Convolutional Inner Encoders.

Consider the inner code and its impact on the exponent of N in Equation (23). For a nonrecursive inner encoder, we have $n_M^i = l$. In fact, every input sequence with weight l generates a finite-weight error event, so that an input sequence with weight l will generate, at most, l error events corresponding to the concatenation of l error events of input weight l. Since the uniform interleaver generates all possible permutations of its input sequences, this event will certainly occur. Thus, from Equation (23) we have

$$\alpha_M = n_M^o - l \geq 0$$

and interleaving gain is not allowed. This conclusion holds true for both SMCCC employing a nonrecursive inner encoder and for all SMCBCs, since block codes have codewords corresponding to input words with weight equal to l. For those SMCCCs, we always have, for some h, coefficients of the exponential in h of Expression (16) that increase with N, and this explains the divergence of the bound arising, for each E_b/N_0, when the coefficients increasing with N become dominant.

1.2.2.2. Recursive Inner Encoders.

For recursive convolutional encoders, the minimum weight of input sequences generating error events is *2*. As a consequence, an input sequence of weight l can generate at most $\left\lfloor \frac{l}{2} \right\rfloor$ error events.

Assuming that the inner encoder of the SMCCC is recursive, the maximum exponent of N in Equation (23) becomes

$$\alpha_M = \max_{w,l} \left\{ n_M^o + \left\lfloor \frac{l}{2} \right\rfloor - l - 1 \right\} = \max_{w,l} \left\{ n_M^o - \left\lfloor \frac{l+1}{2} \right\rfloor - 1 \right\} \quad (24)$$

The maximization involves l and w, since n_M^o depends on both quantities. In fact, remembering the definition of n_M^o as the maximum number of concatenated error events of codewords of the outer code with weight l generated by input words of weight w, it is straightforward, as in Equation (19), to obtain

$$n_M^o \leq \left\lfloor \frac{l}{d_f^o} \right\rfloor \quad (25)$$

Substituting now the last inequality, Equation (25), into Equation (24) yields

$$\alpha_M = \max_{l} \left\{ \left\lfloor \frac{l}{d_f^o} \right\rfloor - \left\lfloor \frac{l+1}{2} \right\rfloor - 1 \right\} \quad (26)$$

To perform the maximization of the right-hand side (RHS) of Expression (26), consider first the case of

$$l = q\, d_f^o$$

where q is an integer, so that

$$\alpha_M \leq \max_{q} \left\{ q - \left\lfloor \frac{q d_f^o + 1}{2} \right\rfloor - 1 \right\} \quad (27)$$

The RHS of Expression (27) is maximized, for $2 \leq d_f^o$, by choosing $q = 1$. On the other hand, for

$$q\, d_f^o \leq l < (q+1) d_f^o$$

the most favorable case is $l = q d_f^o$, which leads us again to the previously discussed situation. Thus, the maximization requires $l = d_f^o$. For this value, on the other hand, we have, from Equation (25), $n_M^o \leq 1$, and the inequality becomes an equality if $w \in W_f$, where W_f is the set of input weights w that generates codewords of the outer code with weight $l = d_f^o$. In conclusion, the largest exponent of n is given by

$$\alpha_M = - \left\lfloor \frac{d_f^o + 1}{2} \right\rfloor \quad (28)$$

The value of α_M in Equation (28) shows that the exponents of N in Expression (16) are always negative integers. Thus, for all h, the coefficients of the exponents in h decrease with N, and we always have an interleaver gain. Denoting by $d'_{f,eff}$ the minimum weight of codewords of the inner code generated by weight-2 input sequences, we obtain a different weight $h(\alpha_M)$ for even and odd values of d_f^o. For even d_f^o, the weight $h(\alpha_M)$ associated to the highest exponent of N is given by

$$h(\alpha_M) = \frac{d_f^o \; d'_{f,eff}}{2}$$

Since it is the weight of an inner codeword that concatenates $d_f^o/2$ error events with weight $d_{f,eff}^i$. Substituting the exponent α_M into Expression (16), approximated by only the term of the summation in h corresponding to $h=h(\alpha_M)$, yields

$$\lim_{N\to\infty} P_b(e) \sim \leq B_{even} N^{\frac{d_f^o}{2}} e^{-\left(\frac{d_f^o d_{f,eff}^i R_c E_b}{2 N_o}\right)} \quad (29a)$$

where

$$B_{even} = \frac{d_f^o!}{k p^{\frac{d_f^o}{2}} (\frac{d_f^o}{2})!} \sum_{w \in W_f} w A_{w,d_f^o,1}^o \leq w_{M,f} N_f^o \frac{d_f^o!}{k p^{\frac{d_f^o}{2}} (\frac{d_f^o}{2})!} \quad (29b)$$

In Equation (29b), $w_{M,f}$ is the maximum input weight yielding outer codewords with weight equal to d_f^o, and N_f^o is the number of such codewords.

For d_f^o odd, the value of $h(\alpha_M)$ is given by

$$h(\alpha_M) = \frac{(d_f^o - 3) d_{f,eff}^i}{2} + h_m^{(3)} \quad (30)$$

where $h_m^{(3)}$ is the minimum weight of sequences of the inner code generated by a weight-3 input sequence. In this case, in fact, we have $n_M^i = \frac{d_f^o - 1}{2}$ concatenated error events, of which $n_M^i - 1$ are generated by weight-2 input sequences and one is generated by a weight-3 input sequence.

Thus, substituting the exponent α_M into Expression (16) approximated by keeping only the term of the summation in h corresponding to $h = h(\alpha_M)$ yields

$$\lim_{N\to\infty} P_b(e) \sim\leq B_{odd} N^{\frac{d_f^o-1}{2}} e^{-\left(\left[\frac{(d_f^o-3) d_{f,eff}^i}{2} + h_m^{(3)}\right] R_c \frac{E_b}{N_o}\right)} \quad (31)$$

where

$$B_{odd} = \frac{d_f^o!}{k p^{\frac{(d_f^o-1)}{2}} \left[\frac{(d_f^o-3)}{2}\right]!} \sum_{w \in W_f} w A_{w,d_f^o,1}^o \leq w_{M,f} N_f^o \frac{d_f^o!}{k p^{\frac{d_f^o-1}{2}} \left[\frac{(d_f^o-3)}{2}\right]!} \quad (32)$$

In cases of d_f^o both even and odd, we can draw from Expressions (29) and (31) a few important design considerations, as follows:

(1) In contrast with the case of block codes and nonrecursive convolutional inner encoders, the use of a recursive convolutional inner encoder always yields an interleaver gain. As a consequence, the first design rule states that the inner encoder must be a convolutional recursive encoder.

(2) The coefficient $h(\alpha_M)$ that multiplies the signal-to-noise ratio E_b/N_0 in Expression (16) increases for increasing values of $d_{f,eff}^i$. Thus, we deduce that the effective free distance of the inner code must be maximized. Both this and the previous design rule also had been stated for PMCCCs. As a consequence, the recursive convolutional encoders optimized for use in PMCCCs can be employed altogether as inner CC in SMCCCs. When d_f^o is odd, for special cases it is possible to increase $h(\alpha_M)$ and h_m further by choosing the feedback polynomial of the inner code to have a factor $(1+D)$, yielding $h_m^{(3)} = \infty$. Note that there are other feedback polynomials such as $(1 + D + D^2 + D^3 + D^4)$ or $(1 + D + D^2 + D^3 + D^4 + D^5 + D^6)$ yielding $h_m^{(3)} = \infty$.

(3) The interleaver gain is equal to $N^{-\frac{d_f^o}{2}}$ for even values of d_f^o and to $N^{-\frac{d_f^o-1}{2}}$ for odd values of d_f^o. As a consequence, we should choose, compatibly with the desired rate R_c of the SMCCC, an outer code with a large and, possibly, an odd value of the free distance.

(4) As to other outer code parameters, N_f^o and $w_{M,f}$ should be minimized. In other words, we should have the minimum number of input sequences generating free distance error events of the outer code, and their input weights should be minimized. Since nonrecursive encoders have error events with $w = 1$ and, in general, less input errors associated with error events at free distance, it can be convenient to choose as an outer code a nonrecursive encoder with minimum N_f^o and $w_{M,f}$.

1.2.2.3. Examples Confirming the Design Rules

To confirm the design rules obtained asymptotically,(i.e., for large signal-to-noise ratios and large interleaver lengths *N)* we evaluate the upper bound, Expression (16), to the bit-error probability for several block and convolutional SMCCs with different interleaver lengths, and compare their performances with those predicted by the design guidelines.

1.2.2.3.1. Serially Multiple Concatenated Block Codes.

We consider three different SMCBCs obtained as follows:

a) The first is the *(7m, 3m, N)* SMCBC;

b) The second is a *(15m, 4m, N)* SMCBC using as outer code a *(5, 4)* parity-check code and as inner code a *(15, 5)* Bose-Chaudhuri-Hocquenghem (BCH) code;

c) The third is a *(15m, 4m, N)* SMCBC using as outer code a *(7, 4)* Hamming code and as inner code a *(15, 7)* BCH code.

Note that the second and third SMCBCs have the same rate, *4/15*. The outer, inner, and SMCBC code parameters introduced in the design analysis are listed in Table 2.

In Figures 18, 19 and 20, we plot the bit-error probability bounds for SMCBCs 1, 2 and 3 of Table 2. Code SMCBC1 has $d^o_f = 2$; thus, from Equation (21), we expect an interleaver gain going as *N-1*. This is confirmed by the curves of Figure 18, which, for a fixed and sufficiently large signal-to-noise ratio, show a decrease in P_b *(e)* of a factor of 10 when *N* passes from 4 to 40, from 40 to 400, and from 400 to 4000. Moreover, from Expression (22), we expect, in each curve for ln P_b *(e)*, a slope with E_b/N_0 as $-h_m R_c$. From Table 2, we know that $R_c = 3/7$, $-h_m = 3$, so that $P_b(e)$ should decrease by a factor of $e^{h_m R_c} = 3.6$ when the signal-to-noise ratio increases by *1* (not in dB). This behavior fully agrees with the curves of Figure 18. Finally, the curves of Figure 18 show a divergence of the bound at lower E_b/N_0 for increasing *N*. This is due to coefficients of terms with $h > h_m$ in Expression (16) that increase with *N* and whose influence becomes more important for larger *N*.

Table 2. Design parameters of CCs and SMCBCs for three serially concatenated block codes.

	Outer code			Inner code		SMCBC			
Code	Code type	w^o_m	d^o_f	Code type	w^i_m	D_{if}	$d_{if,eff}$	h_m	$\alpha(h_m)$
SMCBC1	Parity check (4,3)	1	2	Hamming (7,4)	1	3	3	3	-1
SMCBC2	Parity check (5,4)	1	2	BCH (15,5)	1	7	7	7	-1
SMCBC3	Hamming (7,4)	1	3	BCH (15,7)	1	5	5	5	-2

Code SMCBC2 has $d^o_f = 2$; thus, from Equation (21), we expect the same interleaver gain as for SMCBC1, i.e., *N -1*. This is confirmed by the curves of Figure 19. This code, however, has a larger minimum distance $h_m = 7$, and a rate $R_c = 4/15$. Thus, we expect a steeper descent of $P_b(e)$ with E_b/N_0. More precisely, we expect a decrease by a factor of *6.5* when the

signal-to-noise ratio increases by *1*. This, too, is confirmed by the curves, which also show the bound divergence predicted in our analysis.

Code SMCBC3 has $d^o_f = 3$; thus, from Equation (21), we expect a larger interleaver gain than for SMCBC1 and SMCBC2, i.e., N^{-2}. This is confirmed by the curves of Figure 20, which, for a fixed and sufficiently large signal-to-noise ratio, show a decrease in $P_b(e)$ of a factor of 100 when N passes from 7 to 70, from 70 to 700, and from 700 to 7000. This code has a minimum distance $h_m=5$ and a rate $R_c = 4/15$, which means a descent of $P_b(e)$ with E_b/N_0 by a factor of *3.8* when the signal-to-noise ratio increases by *1*. This, too, is confirmed by the curves. As to the bound divergence, we notice a slightly different behavior with respect to previous cases. The curve with $N = 7000$, in fact, denotes a strong influence of coefficients increasing with N for E_b/N_0 lower than *7*.

1.2.2.3.2. Serially Multiple Concatenated Convolutional Codes.

We consider four different SMCCCs obtained as follows: The first, SMCCC1, is a *(3,1,N)* SMCCC, using as outer code a four-state (2,1) recursive, systematic convolutional encoder and as inner code a four-state (3,2) recursive, systematic convolutional encoder. The second, SMCCC2, is a (3,1,N) SMCCC, using as outer code the same four-state (2,1) recursive, systematic convolutional encoder as SMCCC1, and as inner code a four-state (3,2) nonrecursive convolutional encoder. The third, SMCCC3, is a (3,1,N) SMCCC, using as outer code a four-state (2,1) nonrecursive, convolutional encoder, and as inner code the same four-state (3,2) recursive, systematic convolutional encoder as SMCCC1. Finally, the fourth, SMCCC4, is a (6,2,N) SMCCC using as outer code a four-state (3,2) nonrecursive convolutional encoder, and as inner code a four-state (6,3) recursive, systematic convolutional encoder obtained by using three times the four-state (2,1) recursive, systematic convolutional encoders in Table 2.

The outer, inner, and SMCCC code parameters introduced in the design analysis are listed in Table 3. In this table, the CCs are identified through the descriptions of Table 2. In Figures 21, 22, 23, and 24, we plot the bit-error probability bounds for SMCCCs 1,2,3, and 4 of Table 3, with input information block lengths $R^o_c N = 100, 200, 300, 400, 500$, and *1000*.

Consider first the SMCCCs employing as inner CCs recursive, convolutional encoders. They are SMCCC1, SMCCC3, and SMCCC4. Code SMCCC1 has $d^o_f = 5$; thus, from Expression (31), we expect an interleaver gain behaving as N^{-3}. This is fully confirmed by the curves of Figure 21, which, for a fixed and sufficiently large signal-to-noise ratio, show a decrease in $P_b(e)$ of a factor of *1000* when N passes from *200* to *2000*. For an even more accurate confirmation, one can compare the interleaver gain for every pair of curves in the Figure 21. Moreover, from Expression (31), we expect in each curve for $\ln P_b(e)$ a slope with E_b/N_0 as $-h(\alpha_M)R_c$. From Table 3, we know that $R_c=1/3$ and $h(\alpha_M)=7$,so that $P_b(e)$ should decrease by a factor of 10.3 when the signal-to-noise ratio increases by *1*. This behavior fully agrees with the curves of Figure 21. Finally, the curves of Figure 21 do not show a divergence of the bound at lower E_b/N_0 for increasing N. This is due to the choice of a recursive encoder for the inner code, which guarantees that all coefficients $\alpha(h)$ decrease with N.

Table 3. Design parameters of MCCs and SMCCCs for four SMCCCs. Outer code Inner code SMCCC

Code	Code type	w^o_m	d^o_f	Code type	w^i_m	d_{if}	$d_{if,eff}$	h_m	$\alpha(h_m)$	$h(\alpha_M)$	α_M
SMCCC 1	Rate 1/2 recursive	2	5	Rate 2/3 recursive	2	3	4	5	-4	7	-3
SMCCC 2	Rate 1/2 recursive	2	5	Rate 2/3 nonrecursive	1	3	4	5	-4	--	--
SMCCC 3	Rate 1/2 nonrecursive	1	5	Rate 2/3 recursive	2	3	4	5	-4	7	-3
SMCCC 4	Rate 2/3 nonrecursive	1	3	Rate 1/2 recursive	2	5	6	5	-2	5	-2

Code SMCCC3 differs from SMCCC1 only in the choice of a nonrecursive outer encoder, which is a four-state encoder (see Tables 2 and 3) with the same d^o_f as for SMCCC1, but with $w^o_m = 1$ instead of $w^o_m = 2$.

From the design conclusions, we expect a slightly better behavior from this SMCCC. This is confirmed by the performance curves of Figure 23, which present the same interleaver gain and slope as those of SMCCC1 but have a slightly lower $P_b(e)$ (the curves for SMCCC3 are translated versions of those of SMCCC1 by 0.1 dB).

Code SMCCC4 employs the same CCs as SMCCC2 but reverses their order. It uses as outer code a rate 2/3 nonrecursive convolutional encoder, and as inner code, a rate 1/2 recursive convolutional encoder. As a consequence, it has a lower $d^o_f = 3$ and a higher $\alpha_M = -2$. Thus, from Expression (31), we expect a lower interleaver gain than for SMCCC1 and SMCCC3 as N^{-2}. This is confirmed by the curves of Figure 24, which, for a fixed and sufficiently large signal-to-noise ratio, show a decrease in $P_b(e)$ of a factor of 100 when N passes from 150 to 1500. As to the slope with E_b/N_0, this code has the same $-h(\alpha_M)R_c$ as SMCCC1 and SMCCC3 and, thus, the same slope. On the whole, SMCCC4 loses more than 2 dB in coding gain with respect to SMCCC3. This result confirms the design rule suggesting the choice of an outer code with d^o_f as large as possible.

Finally, let us consider code SMCCC2, which differs from SMCCC1 in the choice of a nonrecursive inner encoder, with the same parameters but with the crucial difference of $w^i_m = 1$. Its bit-error probability curves are shown in Figure 22. We see, in fact, that for low signal-to-noise ratios, say below 3, no interleaver gain is obtained. This is because the performance is dominated by the exponent $h(\alpha_M)$, whose coefficient increases with N. On the other hand, for larger signal-to-noise ratios, where the dominant contribution to $P_b(e)$ is the exponent with the lowest value of h_m, the interleaver gain makes its appearance. From Expression (22), we foresee a gain as N^{-4}, meaning four orders of magnitude for N passing from 100 to 1000. Curves in Figure 22 show a smaller gain (slightly higher than 1/1000), which is, on the other hand, rapidly increasing with E_b/N_0.

1.3. Parallel Multiple Concatenated Convolutional Codes

The concept of Parallel Multiple Concatenated Convolutional Code (PMCCC) utilizes a soft-output decoding and an iterative decoding. We present two versions of a simplified maximum a posteriori (MAP) decoding algorithm. The algorithms work in a sliding window form (like the Viterbi algorithm) and can thus be used to decode continuously transmitted sequences obtained by PMCCC, without requiring code trellis termination. A heuristic explanation is also given of how to embed the maximum a posteriori algorithms into the iterative decoding of PMCCC. The performances of the two algorithms are compared on the basis of a powerful rate 1/3 PMCCC. Basic circuits to implement the simplified a posteriori decoding algorithm using lookup tables, and two further approximations (linear and threshold), with a very small penalty, to eliminate the need for lookup tables are proposed.

The broad framework of this analysis encompasses digital transmission systems where the received signal is a sequence of waveforms whose correlation extends well beyond T, the signaling period. There can be many reasons for this correlation, such as coding, intersymbol interference (ISI), or correlated fading. The optimum receiver in such situations cannot perform its decisions on a symbol-by-symbol basis, so that deciding on a particular information symbol u_k involves processing a portion of the received signal T_d seconds long, with $T_d > T$. The decision rule can be either optimum with respect to a sequence of symbols, $u_k^n = (u_k, u_{k+1}, \cdots, u_{k+n-1})$, or with respect to the individual symbol, u_k.

The most widely applied algorithm for the first kind of decision rule is the Viterbi algorithm. In its optimum formulation, it would require waiting for decisions until the whole sequence have been received. In practical implementations, this drawback is overcome by anticipating decisions (single or in batches) on a regular basis with a fixed delay, D. Choice of D as five to six times the memory of the received data is widely recognized as a good compromise between performance, complexity, and decision delay.

Optimum symbol decision algorithms must base their decisions on the maximum a posteriori (MAP) probability. They have been known since the early seventies, although much less popular than the Viterbi algorithm and almost never applied in practical systems. There is a very good reason for this neglect in that they yield performance in terms of symbol error probability only slightly superior to the Viterbi algorithm, yet they present a much higher conceptual complexity. Only recently, the interest in these algorithms has seen a revival in connection with the problem of decoding concatenated coding schemes. Concatenated coding schemes (a class in which we include product codes, multilevel codes, generalized concatenated codes, and serial and parallel concatenated codes) were proposed as a means of achieving large coding gains by combining two or more relatively simple "constituent" codes. The resulting concatenated coding scheme is a powerful code endowed with a structure that permits an easy decoding, like "stage decoding" or "iterated stage decoding".

To work properly, all these decoding algorithms cannot limit themselves to passing the symbols decoded by the inner decoder to the outer decoder. They need to exchange some kind of soft information. The optimum output of the inner decoder should be in the form of the sequence of the probability distributions over the inner code alphabet conditioned on the received signal, the a posteriori probability (APP) distribution. There have been several attempts to achieve, or at least to approach, this goal. Some of them are based on modifications of the Viterbi algorithm so as to obtain, at the decoder output, in addition to the "hard"-decoded symbols, some reliability information. This has led to the concept of "augmented-output" or the list-decoding Viterbi algorithm, and to the soft-output Viterbi algorithm (SOVA). These solutions are clearly sub-optimal, as they are unable to supply the required APP. A different approach consisted in revisiting the original symbol MAP decoding algorithms with the aim of simplifying them to a form suitable for implementation. Figure 25 shows a PMCCC whose encoder is formed by two (or more) constituent systematic encoders joined through an interleaver. The input information bits feed the first encoder and, after having been interleaved by the interleaver, enter the second encoder. The codeword of the PMCCC comprises of the input bits to the first encoder followed by the parity check bits of both encoders. Generalizations to more than one interleaver are possible and fruitful.

The sub-optimal iterative decoder is modular and comprises of a number of equal component blocks formed by concatenating soft decoders of the constituent codes (CC) separated by the interleavers used at the encoder side. By increasing the number of decoding modules and, thus, the number of decoding iterations, bit-error probabilities as low as 10^{-5} at E_b/N_0 =0.0 dB for rate 1/4 PMCCC have been shown by simulation.

We will describe two versions of a simplified MAP decoding algorithm that can be used as building blocks of the iterative decoder to decode PMCCCs. A distinctive feature of the algorithms is that they work in a "sliding window" form, like the Viterbi algorithm, and thus can be used to decode "continuously transmitted" PMCCCs, without requiring trellis termination and a block-equivalent structure of the code.

The final aim is to find suitable soft-output decoding algorithms for iterated staged decoding of PMCCC employed in a continuous transmission.

We will refer to the transmission system of Figure 26. The information sequence u, composed of symbols drawn from an alphabet $U=\{u_1,...,u_M\}$ and emitted by the source, enter an encoder that generates code sequences c. Both source and code sequences are defined over a time index set K (a finite or infinite set of integers). Denoting the code alphabet $C=\{c_1,...,c_M\}$, the code C can be written as a subset of the Cartesian product of C by itself K times, i.e., $C \subseteq C^K$. The code symbols c_k (the index k will refer to time) enter the modulator, which performs a one-to-one mapping of them with its signals, or channel input symbols x_k, belonging to the set $X=\{x_1,...,x_M\}$.

The channel symbols x_k are transmitted over a stationary memoryless channel with output symbols y_k. The channel is characterized by the transitions probability distribution (discrete or continuous, according to the channel model) $P(y|x)$. The channel output sequence is fed to the symbol-by-symbol soft-output demodulator, which produces a sequence of probability distributions $\gamma_k(c)$ over C conditioned on the received signal, according to the memoryless transformation.

$$\gamma_k(c) = P(x_k = x(c), y_k) = P(y_k | x_k = x(c)) \; P_k(c) = \gamma_k(x) \qquad (33)$$

where we have assumed to know the sequence of the a priori probability distributions of the channel input symbols $(P_k(x): k \in K)$ and made use of the one-to-one mapping $C \rightarrow X$.

The sequence of probability distributions $\gamma_k(c)$ obtained by the modulator on a symbol-by-symbol basis is then supplied to the soft-output symbol decoder, which processes the distributions in order to obtain the probability distributions $P_k(u|y)$. They are defined as

$$P_k(u|y) = P(u_k = u|y) \qquad (34)$$

The probability distributions $P_k(u|y)$ are referred to in the literature as symbol-by-symbol a posteriori probabilities (APP) and represent the optimum symbol-by-symbol soft output.

From here on, we will limit ourselves to the case of time-invariant convolutional codes with N states, use the following notations with reference to Figure 27, and assume that the (integer) time instant we are interested in is the k^{th}:

(1) S_i is the generic state at time k, belonging to the set $S = \{S_1, \cdots, S_N\}$.

(2) $S_i^-(u')$ is one of the precursors of S_i, and precisely the one defined by the information symbol u' emitted during the transition $S_i^-(u') \rightarrow S_i^2$

(3) $S_i^+(u)$ is one of the successors of S_i, and precisely the one defined by the information symbol u emitted during the transition $S_i \rightarrow S_i^+(u)$.

(4) To each transition in the trellis, a signal x is associated, which depends on the state from which the transition originates and on the information symbol u determining that transition. When necessary, we will make this dependence explicit by writing $x(u', S_i)$ when the transition ends in S_i and $x(S_i, u)$ when the transition originates from S_i.

1.3.1 The BCJR Algorithm

The BCJR is the optimum algorithm to produce the sequence of APP. We consider first the original version of the algorithm, which applies to the case of a finite index set $K = \{1,...,n\}$ and requires the knowledge of the whole received sequence $y = (y_1,...,y_n)$ to work. In the following, the notations u, c, x, and y will refer to sequences n-symbols long, and the integer time variable k will assume the values $1, \cdots, n$. As for the previous assumption, the encoder admits a trellis representation with N states, so that the code sequences c (and the corresponding transmitted signal sequences x) can be represented as paths in the trellis and uniquely associated with a state sequence $s = (s_0,...,s_N)$ whose first and last states, s_0 and s_N, are assumed to be known by the decoder.

Defining the a posteriori transition probabilities from state S_i at time k as

$$\sigma_k(S_i, u) = P(u_k = u, s_{k-1} = S_i \mid y) \qquad (35)$$

The APP $P(u|y)$ we want to compute can be obtained as

$$P_k(u|y) = \sum_{S_i} \sigma_k(S_i, u) \qquad (36)$$

Thus, the problem of evaluating the APP is equivalent to that of obtaining the a posteriori transition probabilities defined in Equation (35). The APP can be computed as

$$\sigma_k(S_i, u) = h_\sigma \; \alpha_{k-1}(S_i) \; \gamma_k(x(S_i, u)) \; \beta_k(S_i^-(u)) \qquad (37)$$

where:

- h_σ is such that $\sum_{S_i, u} \sigma_k(S_i, u) = 1$
- $\gamma_k(x(S_i, u))$ are the joint probabilities already defined in Equation (33), i.e.,

$$\gamma_k(x) = P(y_k, x_k = x) = P(y_k | x_k = x) \, P(x_k = x) \qquad (38)$$

The γs can be calculated from the knowledge of the a priori probabilities of the channel input symbols x and of the transition probabilities of the channel $P(y_k|x_k=x)$. For each time k, there are M different values of γ to be computed, which are then associated to the trellis transitions to form a sort of branch metrics. This information is provided by the symbol-by-symbol soft-output demodulator.

- $\alpha_k(S_i)$ are the probabilities of the states of the trellis at time k conditioned on the past received signals, namely,

$$\alpha_k(S_i) = P(s_k = S_i \mid y_1^k) \qquad (39)$$

where y_1^k denotes the sequence $y_1,\ldots,y_k$. They can be obtained by the forward recursion:

$$\alpha_k(S_i) = h_\alpha \sum_u \alpha_{k-1}(S_i^-(u))\, \gamma_k(x(u,S_i)) \qquad (40)$$

with h_α a constant determined through the constraint

$$\sum_{S_i} \alpha_k(S_i) = 1$$

and where the recursion is initialized as

$$\alpha_0(S_i) = \left\{ \begin{array}{ll} 1 & \text{if } S_i = s_0 \\ 0 & \text{otherwise} \end{array} \right\} \qquad (41)$$

- $\beta_k(S_i)$ are the probabilities of the trellis states at time k conditioned on the future received signals $P(s_k = S_i \mid y_{k+1}^n)$. They can be obtained by the backward recursion

$$\beta_k(S_i) = h_\beta \sum_u \beta_{k+1}(S_i^+(u))\, \gamma_{k+1}(x(S_i,u)) \qquad (42)$$

with h_β a constant obtainable through the constraint

$$\sum_{S_i} \beta_k(S_i) = 1$$

and where the recursion is initialized as

$$\beta_n(S_i) = \left\{ \begin{array}{ll} 1 & \text{if } S_i = s_n \\ 0 & \text{otherwise} \end{array} \right\} \qquad (43)$$

We can now formulate the BCJR algorithm by the following steps:

(1) Initialize α_0 and β_n according to Equations (41) and (43).

(2) As soon as each term y_k of the sequence y is received, the demodulator supplies to the decoder the "branch metrics" γ_k of Equation (38), and the decoder computes the probabilities α_k according to Equation (40). The obtained values of $\alpha_k(S_i)$ as well as the γ_k are stored for all k, S_i, and x.

(3) When the entire sequence y has been received, the decoder recursively computes the probabilities β_k according to the recursion of Equation (42) and uses them together with the stored α's and β's to compute the a posteriori transition probabilities $\alpha_k(S_i, u)$ according to Equation (37) and, finally, the APP $P_k(u|y)$ from Equation (36).

1.3.2 The Sliding Window BCJR (SW-BCJR)

The BCJR algorithm requires that the whole sequence have been received before starting the decoding process. In this aspect, it is similar to the Viterbi algorithm in its optimum version. To apply it in a PMCCC, we need to subdivide the information sequence into blocks, decode them by terminating the trellises of both CCs, and then decode the received sequence block by block. Beyond the rigidity, this solution also reduces the overall code rate.

A more flexible decoding strategy is offered by a modification of the BCJR algorithm in which the decoder operates on a fixed memory span, and decisions are forced with a given delay D. We call this new, and sub-optimal, algorithm the sliding window BCJR (SW-BCJR) algorithm. We will describe two versions of the sliding window BCJR algorithm that differ in the way they overcome the problem of initializing the backward recursion without having to wait for the entire sequence. We will describe the two algorithms using the previous step description suitably modified. Of the previous assumptions, we retain only that of the knowledge of the initial state s_0, and thus assume the transmission of semi-infinite code sequences, where the time span K ranges from 1 to ∞.

1.3.2.1. The First Version of the Sliding Window BCJR Algorithm (SW1-BCJR)

Here are the steps:

(1) Initialize α_0 according to Equation (41).

(2) Forward recursion at time k: Upon receiving y_k, the demodulator supplies to the decoder the M distinct branch metrics, and the decoder computes the probabilities $\alpha_k(S_i)$ according to Equations (38) and (40). The obtained values of $\alpha_k(S_i)$ are stored for all S_i, as well as the $\alpha_k(x)$.

(3) Initialization of the backward recursion $(k>D)$:

$$\beta_k(S_j) = \alpha_k(S_j) \quad \forall S_j \qquad (44)$$

(4) Backward recursion: It is performed according to Equation (42) from time $k-1$ back to time $k-D$.

(5) The a posteriori transition probabilities at time $k-D$ are computed according to

$$\alpha_{k-D}(S_i,u) = h_\sigma \; \alpha_{k-D-1}(S_i) \; \gamma_{k-D}(x(S_i,u)) \; \beta_{k-D}(S_i^+(u)) \qquad (45)$$

(6) The APP at time $k-D$ is computed as

$$P_{k-D}(u|y) = \sum_{S_i} \sigma_{k-D}(S_i,u) \qquad (46)$$

For a convolutional code with parameters (k_0, n_0), number of states N, and cardinality of the code alphabet, $M = 2^{n_0}$, the SW1-BCJR algorithm requires storage of N ×D values of α's and M ×D values of the probabilities $\gamma_k(x)$ generated by the soft demodulator. Moreover, to update the α's and β's for each time instant, the algorithm needs to perform $M \times 2^{k_0}$ multiplications and N additions of 2^{k_0} numbers. To output the set of APP at each time instant, we need a D-times long backward recursion. Thus, the computational complexity requires *overall* $(D+1)\,M \times 2^{k_0}$ multiplications and $(D+1)\,M$ additions of 2^{k_0} numbers each

As a comparison, the Viterbi algorithm would require, in the same situation, $M \times 2^{k_0}$ additions and $M \times 2^{k_0}$-way comparisons, plus the trace-back operations, to get the decoded bits.

1.3.2.2 The Second, Simplified Version of the Sliding Window BCJR Algorithm (SW2-BCJR)

A simplification of the sliding window BCJR that significantly reduces the memory requirements comprises of the following steps:

(1) Initialize α_0 according to Equation (41).

(2) Forward recursion $(k > D)$: If $k > D$, the probabilities $\alpha_{k-D-1}(S_i)$ are computed according to Equation (40).

(3) Initialization of the backward recursion $(k > D)$:

$$\beta_k(S_j) = \frac{1}{N} \quad \forall S_j \qquad (47)$$

(4) Backward recursion $(k > D)$: It is performed according to Equation (42) from time $k-1$ back to time $k-D$.

(5) The a posteriori transition probabilities at time $k-D$ are computed according to

$$\alpha_{k-D}(S_i,u) = h_\sigma \; \alpha_{k-D-1}(S_i) \; \gamma_{k-D}(x(S_i,u)) \; \beta_{k-D}(S_i^*(u)) \qquad (48)$$

(6) The APP at time $k - D$ is computed as

$$P_{k-D}(u|y) = \sum_{S_i} \sigma_{k-D}(S_i,u) \qquad (49)$$

This version of the sliding window BCJR algorithm does not require storage of the $N \times D$ values of α's as they are updated with a delay of D steps. As a consequence, only N values of α's and $M \times D$ values of the probabilities $\gamma_k(x)$ generated by the soft demodulator must be stored. The computational complexity is the same as the previous version of the algorithm. However, since the initialization of the β recursion is less accurate, a larger value of D should be set in order to obtain the same accuracy on the output values $P_{k-D}(u|y)$. This observation will receive quantitative evidence in the section devoted to simulation results.

1.3.3 Additive Algorithms

1.3.3.1 The Log-BCJR

The BCJR algorithm and its sliding window versions have been stated in multiplicative form. Owing to the monotonicity of the logarithm function, they can be converted into an additive form by conversion to logarithms. Let us define the following logarithmic quantities:

$$\Gamma_k(x) = \log [\gamma(x)]$$

$$A_k(S_i) = \log [\alpha_k(S_i)]$$

$$B_k(S_i) = \log [\beta_k(S_i)]$$

$$\Sigma_k(S_i,u) = \log [\sigma_k(S_i,u)]$$

These definitions lead to the following A and B recursions, derived from Equations (40), (42), and (37):

$$A_k(S_i) = \log \left[\sum_u e^{\left(A_{k-1}(S_i^*(u)) + \Gamma_k(x(u,S_i))\right)} \right] + H_A \qquad (50)$$

$$B_k(S_i) = \log \left[\sum_u e^{\left(\Gamma_{k+1}(S_i(u)) + B_{k+1}(S_i^*(u))\right)} \right] + H_B \qquad (51)$$

$$\Sigma_k(S_i,u) = A_{k-1}(S_i) + \Gamma_k(x(S_i,u)) + B_k(S_i^*(u)) + H_\Sigma \qquad (52)$$

with the following initializations:

$$A_0(S_i) = \left\{ \begin{matrix} 1 & if\ S_i = s_0 \\ -\infty & otherwise \end{matrix} \right\}$$

$$B_1(S_i) = \left\{ \begin{matrix} 1 & if\ S_i = s_n \\ -\infty & otherwise \end{matrix} \right\}$$

1.3.3.2 Simplified Versions of the Log-BCJR

The problem in the recursions defined for the log-BCJR comprises of the evaluation of the logarithm of a sum of exponential:

$$\log \left[\sum_i e^{A_i} \right]$$

An accurate estimate of this expression can be obtained by extracting the term with the highest exponential,

$$A_M = \max_i \{ A_i \}$$

so that

$$\log\left[\sum_i e^{A_i}\right] = A_M + \log\left(1 + \sum_{A_i \neq A_M} e^{(A_i - A_M)}\right) \quad (53)$$

and by computing the second term of the right-hand side (RHS) of Equation (53) using lookup tables.

However, when $A_M >> A_i$, the second term can be neglected. This approximation leads to the additive logarithmic-BCJR (AL-BCJR) algorithm:

$$A_k(S_i) = \max_u \left[A_{k-1}(S_i(u)) + \Gamma_k(x(u,S_i)) \right] + H_A \quad (54)$$

$$B_k(S_i) = \max_u \left[B_{k+1}(S_i^+(u)) + \Gamma_{k+1}(S_i^+(u)) \right] + H_B \quad (55)$$

$$\Sigma_k(S_i,u) = A_{k-1}(S_i) + \Gamma_k(x(S_i,u)) + B_k(S_i^+(u)) + H_\Sigma \quad (56)$$

with the same initialization of the log-BCJR.

Both versions of the SW-BCJR algorithm described can be used, with obvious modifications, to transform the block log-BCJR and the AL-BCJR into their sliding window versions, leading to the SW-log-BCJR and the SWAL1-BCJR and SWAL2-BCJR algorithms.

1.3.4. Explicit Algorithms for Some Particular Cases

In this section, we will make explicit the quantities considered in the previous algorithms' descriptions by making assumptions on the code type, modulation format, and channel.

1.3.4.1. Rate 1/n Binary Systematic Convolutional Encoder

In this section, we particularize the previous equations in the case of a rate *1/n* binary systematic encoder associated to n binary-pulse amplitude modulation (PAM) signals or binary phase shift keying (PSK) signals.

The channel symbols x and the output symbols from the encoder can be represented as vectors of n binary components:

$$\tilde{c} = [c_1,\ldots,c_n] \quad c_i \in \{0,1\}$$

$$\tilde{x} = [x_1,\ldots,x_n] \quad x_i \in \{A,-A\}$$

$$\tilde{x}_k = [x_{k1},\ldots,x_{kn}]$$

$$\tilde{y}_k = [y_{k1},\ldots,y_{kn}]$$

where the notations have been modified to show the vector nature of the symbols. The joint probabilities $\gamma_k(\tilde{x})$, over a memoryless channel, can be split as

$$\gamma_k(\tilde{x}) = \prod_{m=1}^{n} P(y_{km}|x_{km} = x_m)\, P(x_{km} = x_k) \quad (57)$$

Since in this case the encoded symbols are n-tuple of binary symbols, it is useful to redefine the input probabilities, γ, in terms of the likelihood ratios:

$$\lambda_{km} = \frac{P(y_{km} \mid x_{km} = A)}{P(y_{km} \mid x_{km} = -A)}$$

$$\lambda_{km}^A = \frac{P(x_{km} = A)}{P(x_{km} = -A)}$$

so that, from Equation (57),

$$\gamma_k(\tilde{x}) = \prod_{m=1}^{n} \frac{(\lambda_{km})^{c_m}}{1+\lambda_{km}} \frac{(\lambda_{km}^A)^{c_m}}{1+\lambda_{km}^A} = h_\gamma \prod_{m=1}^{n} [\lambda_{km}\ \lambda_{km}^A]^{c_m}$$

where h_γ takes into account all terms independent of $\tilde{x}$.

The BCJR can be restated as follows:

$$\alpha_k(S_i) = h_\gamma \; h_\alpha \sum_u \alpha_{k-1}(S_i^-(u)) \prod_{m=1}^{n} [\lambda_{km} \; \lambda_{km}^A]^{c_m(u,S_i)} \quad (58)$$

$$\beta_k(S_i) = h_\gamma \; h_\beta \sum_u \beta_{k+1}(S_i^+(u)) \prod_{m=1}^{n} [\lambda_{(k+1)m} \; \lambda_{(k+1)m}^A]^{c_m(S_i,u)} \quad (59)$$

$$\sigma_k(S_i,u) = h_\gamma \; h_\sigma \; \alpha_{k-1}(S_i) \prod_{m=1}^{n} [\lambda_{(k+1)m} \; \lambda_{(k+1)m}^A]^{c_m(u,S_i)} \; \beta_k(S_i^+(u)) \quad (60)$$

whereas its simplification, the AL-BCJR algorithm, becomes

$$A_k(S_i) = \max_u \left\{ A_{k-1}(S_i^-(u)) + \sum_{m=1}^{n} c_m(u,S_i)\,(\Lambda_{km} + \Lambda_{km}^A) \right\} + H_A \quad (61)$$

$$B_k(S_i) = \max_u \left\{ B_{k+1}(S_i^+(u)) + \sum_{m=1}^{n} c_m(S_i,u)\,(\Lambda_{km} + \Lambda_{km}^A) \right\} + H_B \quad (62)$$

$$\Sigma_k(S_i,u) = A_{k-1}(S_i) + \sum_{m=1}^{n} c_m(S_i,u)\,(\Lambda_{km} + \Lambda_{km}^A) + B_k(S_i^+(u)) \quad (63)$$

where Λ stands for the logarithm of the corresponding quantity λ.

1.3.4.2 The Additive White Gaussian Noise Channel

When the channel is an additive white Gaussian noise (AWGN) channel, we obtain the explicit expression of the log-likelihood ratios Λ_{ki} as

$$\Lambda_{ki} = \log\left[\frac{P(y_{ki}|x_{ki} = A)}{P(y_{ki}|x_{ki} = -A}\right] = \log\left[\frac{\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(\frac{1}{2\sigma^2}(y_{ki}-A)^2\right)}}{\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(\frac{1}{2\sigma^2}(y_{ki}+A)^2\right)}}\right] = \frac{2A}{\sigma^2} y_{ki}$$

Hence, the AL-BCJR algorithm assumes the following form:

$$A_k(S_i) = \max_u \left\{ A_{k-1}(S_i^-(u)) + \sum_{m=1}^{n} c_m(u,S_i)\,(\frac{2A}{\sigma^2} y_{km} + \Lambda_{km}^A) \right\} + H_A \quad (64)$$

$$B_k(S_i) = \max_u \left\{ B_{k+1}(S_i^+(u)) + \sum_{m=1}^{n} c_m(S_i,u)\,(\frac{2A}{\sigma^2} y_{km} + \Lambda_{km}^A) \right\} + H_B \quad (65)$$

$$\Sigma_k(S_i,u) = A_{k-1}(S_i) + \sum_{m=1}^{n} c_m(S_i,u)\,(\frac{2A}{\sigma^2} y_{km} + \Lambda_{km}^A) + B_k(S_i^+(u)) \quad (66)$$

We will consider turbo codes with rate 1/2 component convolutional codes transmitted as binary PAM or binary PSK over an AWGN channel.

1.3.5. Iterative Decoding of Parallel Concatenated Convolutional Codes

In this section, we will show how the MAP algorithms previously described can be embedded into the iterative decoding procedure of parallel concatenated codes. We will derive the iterative decoding algorithm through suitable approximations performed on maximum-likelihood decoding. The description will be based on the fairly general parallel concatenated code shown in Figure 28, which employs three encoders and three interleavers (denoted by π in the figure).

Let u_k be the binary random variable taking values in {0,1}, representing the sequence of information bits $u=(u_1,\ldots,u_n)$. The optimum decision algorithm on the k^{th} bit u_k is based on the conditional log-likelihood ratio L_k:

$$L_k = \log\frac{P(u_k=1|y)}{P(u_k=0|y)}$$

$$L_k = \log\frac{\sum_{u,u_k=1} P(y|u)\prod_{j\neq k} P(u_j)}{\sum_{u,u_k=0} P(y|u)\prod_{j\neq k} P(u_j)} + \log\frac{P(u_k=1)}{P(u_k=0)}$$

$$L_k = \log\frac{\sum_{u,u_k=1} P(y|x(u))\prod_{j\neq k} P(u_j)}{\sum_{u,u_k=0} P(y|x(u))\prod_{j\neq k} P(u_j)} + \log\frac{P(u_k=1)}{P(u_k=0)} \quad (67)$$

where, in Equation (67), $P(u_j)$ are the a priori probabilities.

If the rate k_o/n_o constituent code is not equivalent to a punctured rate $1/n'_o$ code (in this case the information send is the information data and one parity bit, the parity bit sent is different every time) or if turbo trellis-coded modulation is used, we can first use the symbol MAP algorithm as described in the previous sections to compute the log-likelihood ratio of a symbol $u=u_1,\ldots,u_{k_o}$, given the observation y as

$$\lambda(u) = \log\frac{P(u|y)}{P(0|y)}$$

where 0 corresponds to the all-zero symbol. Then we obtain the log-likelihood ratios of the j^{th} bit within the symbol by

$$L(u_j) = \log\frac{\sum_{u:u_j=1} e^{\lambda(u)}}{\sum_{u:u_j=0} e^{\lambda(u)}}$$

In this way, the turbo decoder operates on bits, rather than symbol, interleaving is used.

To explain the basic decoding concept, we restrict ourselves to three codes, but extension to several codes is straightforward. In order to simplify the notation, consider the combination of the interleaver and the constituent encoder connected to it as a block code with input u and outputs x_i, $i=0,1,2,3$ ($x_0=u$) and the corresponding received sequences as y_i, $i=0,1,2,3$. The optimum bit decision metric on each bit is (for data with uniform a priori probabilities)

$$L_k = \log\frac{\sum_{u,u_k=1} P(y_0|u)\,P(y_1|u)\,P(y_2|u)\,P(y_3|u)}{\sum_{u,u_k=0} P(y_0|u)\,P(y_1|u)\,P(y_2|u)\,P(y_3|u)} \quad (68)$$

but, in practice, we cannot compute Equation (68) for large n because the permutations π_2, π_3 imply that y_2 and y_3 are no longer simple convolutional encodings of u. Suppose that we evaluate $P(y_i|u), i=0,2,3$ in Equation (68) using Bayes' rule and using the following approximation:

$$P(u|y_i) \approx \prod_{k=1}^{n} \tilde{P}_i(u_k) \quad (69)$$

Note that $P(u|y_i)$ is not separable in general. However, for $i=0$, $P(u|y_0)$ is separable; hence, Equation (69) holds with equality. So we need an algorithm that approximates a nonseparable distribution $P(u|y_i) = P$ with a separable distribution $\prod_{k=1}^{N} \tilde{P}_i(u_k) = Q$. The best approximation can be obtained using the Kullback cross-entropy minimizer, which minimizes the cross-entropy $H(Q,P)=E\{\log(Q/P)\}$ between the input P and the output Q.

The MAP algorithm approximates a nonseparable distribution with a separable one; however it is not clear how good it is compared with the Kullback cross-entropy minimizer. Here we use the MAP algorithm for such an approximation. In the iterative decoding, as the reliability of the $\{u_k\}$ improves. intuitively one expects that the cross-entropy between the input and

the output of the MAP algorithm will decrease, so that the approximation will improve. If such an approximation, i.e., Equation (69), can be obtained, we can use it in Equation (68) for $i=2$ and $i=3$ (by Bayes' rule) to complete the algorithm.

Define $\tilde{L}_{ik}$ by

$$\tilde{P}_i(u_k) = \frac{e^{u_k \tilde{L}_{ik}}}{1 + e^{\tilde{L}_{ik}}} \qquad (70)$$

where $u_k \in \{0,1\}$. To obtain $\{ \tilde{P}_i \}$ or, equivalently $\{ \tilde{L}_{ik} \}$, we use Equations (69) and (70) for $i=0,2,3$ (by Bayes' rule) to express Equation (68) as

$$L_k = f(y_1 , \tilde{L}_0 , \tilde{L}_2 , \tilde{L}_3 , k) + \tilde{L}_{0k} + \tilde{L}_{2k} + \tilde{L}_{3k} \qquad (71)$$

where $\tilde{L}_{0k} = 2Ay_{ok}/\sigma^2$ (for binary modulation) and

$$f(y_1 , \tilde{L}_0 , \tilde{L}_2 , \tilde{L}_3 , k) = \log \frac{\sum_{u,u_k=1} P(y_1|u) \prod_{j \neq k} e^{u_j (\tilde{L}_{0j} + \tilde{L}_{2j} + \tilde{L}_{3j})}}{\sum_{u,u_k=0} P(y_1|u) \prod_{j \neq k} e^{u_j (\tilde{L}_{0j} + \tilde{L}_{2j} + \tilde{L}_{3j})}} \qquad (72)$$

We can use Equations (69) and (70) again, but this time for $i=0,1,3$, to express Equation (68) as

$$L_k = f(y_2 , \tilde{L}_0 , \tilde{L}_1 , \tilde{L}_3 , k) + \tilde{L}_{0k} + \tilde{L}_{1k} + \tilde{L}_{3k} \qquad (73)$$

and similarly,

$$L_k = f(y_3 , \tilde{L}_0 + \tilde{L}_1 , \tilde{L}_2 , k) + \tilde{L}_{0k} + \tilde{L}_{1k} + \tilde{L}_{2k} \qquad (74)$$

A solution to Equations (71), (73), and (74) is

$$\begin{aligned} \tilde{L}_{1k} &= f(y_1 , \tilde{L}_0 , \tilde{L}_2 , \tilde{L}_3 , k) \\ \tilde{L}_{2k} &= f(y_2 , \tilde{L}_0 , \tilde{L}_1 , \tilde{L}_3 , k) \\ \tilde{L}_{3k} &= f(y_3 , \tilde{L}_0 , \tilde{L}_1 , \tilde{L}_2 , k) \end{aligned} \qquad (75)$$

for $k=1,2,\cdots,n$, provided that a solution to Equation (75) does indeed exist. The final decision is then based on

$$L_k = \tilde{L}_{0k} + \tilde{L}_{1k} + \tilde{L}_{2k} + \tilde{L}_{3k} \qquad (76)$$

which is passed through a hard limiter with zero threshold. We attempted to solve the nonlinear equations in Equation (75) for $\tilde{L}_1$, $\tilde{L}_2$, and $\tilde{L}_3$ by using the iterative procedure:

$$\tilde{L}_{1k}^{(m+1)} = \alpha_1^{(m)} f(y_1 , \tilde{L}_0 , \tilde{L}_1^{(m)} , \tilde{L}_2^{(m)} , k) \qquad (77)$$

for $k=1,2,\cdots,n$, iterating on m. Similar recursions hold for $\tilde{L}_{2k}^{(m)}$ and $\tilde{L}_{3k}^{(m)}$.

We start the recursion with the initial condition $\tilde{L}_1^{(0)} = \tilde{L}_2^{(0)} = \tilde{L}_3^{(0)} = \tilde{L}_0$. For the computation of $f(\cdot)$, we can use any MAP algorithm as described in the previous sections, with interleavers (direct and inverse) where needed; call this the basic decoder D_i, $i=1,2,3$. The $\tilde{L}_{ik}^{(m)}$, $i=1,2,3$ represent the extrinsic information. The signal flow graph for extrinsic information is shown in Figure 29, which is a fully connected graph without self-loops. Parallel, serial, or hybrid implementations can be realized based on the signal flow graph (in this figure y_0 is considered as part of y_1). Based on our equations, each node's output is equal to internally generated reliability L minus the sum of all inputs to that node. The BCJR MAP algorithm always starts and ends at the all-zero state since we always terminate the trellis. We assumed $\pi_1=I$ identity; however, any π_1 can be used.

The overall decoder is composed of block decoders D_i connected in parallel, as in Figure 30 (when the switches are in position P), which can be implemented as a pipeline or by feedback. A serial implementation is also shown in Figure 30 (when the switches are in position S. For those applications where the systematic bits are not transmitted or for parallel

concatenated trellis codes with high-level modulation, we should set $\tilde{L}_0 = 0$. Even in the presence of systematic bits, if desired, one can set $\tilde{L}_0 = 0$ and consider y_0 as part of y_1 If the systematic bits are distributed among encoders, we use the same distribution for y_0 among the received observations for MAP decoders.

At this point, further approximation for iterative decoding is possible if one term corresponding to a sequence u dominates other terms in the summation in the numerator and denominator of Equation (72). Then the summations in Equation (72) can be replaced by "maximum" operations with the same indices, i.e., replacing $\sum_{u:uk=i}$ with $max_{u:uk=i}$ for $i = 0,1$. A similar approximation can be used for $\tilde{L}_{2k}$ and $\tilde{L}_{3k}$ in Equation. (75). This sub-optimal decoder then corresponds to an iterative decoder that uses AL-BCJR rather than BCJR decoders. As discussed, such approximations have been used by replacing, "$\sum$" with "*max*" in the log-BCJR algorithm to obtain AL-BCJR. Clearly, all versions of SW-BCJR can replace BCJR (MAP) decoders in Figure 30.

For turbo codes with only two constituent codes, Equation (77) reduces to

$$\tilde{L}_{1k}^{(m+1)} = \alpha_1^{(m)} f(y_1 , \tilde{L}_0 , \tilde{L}_2^{(m)} , k)$$

$$\tilde{L}_{2k}^{(m+1)} = \alpha_2^{(m)} f(y_2 , \tilde{L}_0 , \tilde{L}_1^{(m)} , k)$$

for $k = 1,2,\cdots,n$, and $m = 1,2,\cdots$, where, for each iteration, $\alpha_1^{(m)}$ and $\alpha_2^{(m)}$ can be optimized (simulated annealing) or set to *1* for simplicity. The decoding configuration for two codes is shown in Figure 31. In this special case, since the paths in Figure 31 are disjointed, the decoder structure can be reduced to a serial mode structure if desired. If we optimize $\alpha_1^{(m)}$ and $\alpha_2^{(m)}$, this requires estimates of the variances of $\tilde{L}_{1k}$ and $\tilde{L}_{2k}$ for each iteration in the presence of errors.

In the results presented in the next section, we will use a PMCCC with only two constituent codes.

1.3.6. Simulation Results

In this section, we will present some simulation results obtained applying the iterative decoding algorithm, which, in turn, uses the optimum BCJR and the sub-optimal, but simpler, SWAL2-BCJR as embedded MAP algorithms. All simulations refer to a rate 1/3 PMCCC with two equal, recursive convolutional constituent codes with 16 states and generator matrix

$$G(D) = \left[1 , \frac{1 + D + D^3 + D^4}{1 + D^3 + D^4} \right]$$

and an interleaver of length 16,384, using an S-random permutation with S = 40. Each simulation run examined at least 25,000,000 bits. In Figure 32, we plot the bit-error probabilities as a function of the number of iterations of the decoding procedure using the optimum block BCJR algorithm for various values of the signal-to-noise ratio. It can be seen that the decoding algorithm converges down to bit error rate (BER) = 10^{-5} at signal-to-noise ratios of 0.2 dB with nine iterations. The same curves are plotted in Figure 33 for the case of the sub-optimum SWAL2-BCJR algorithm. In this case, 0.75 dB of signal-to-noise ratio is required for convergence to the same BER and with the same number of iterations.

In Figure 34, the bit-error probability versus the signal-to-noise ratio is plotted for a fixed number (5) of iterations of the decoding algorithm and for both optimum BCJR and SWAL2-BCJR MAP decoding algorithms. It can be seen that the penalty incurred by the sub-optimum algorithm amounts to about 0.5 dB.

Algorithms were of the block type. The penalty is completely attributable to the approximation of the sum of exponentials. To verify this, we have used a SW2-BCJR and compared its results with the optimum block BCJR, obtaining the same results.

Finally, in Figures 35 and 36, we plot the number of iterations needed to obtain a given bit-error probability versus the bit signal-to-noise ratio, for the two algorithms. These curves provide information on the delay incurred to obtain a given reliability as a function of the bit signal-to-noise ratio.

1.3.7 Circuits to Implement the MAP Algorithm for Decoding Rate 1/n Component Codes of a PMCCC

We show the basic circuits required for the implementation of a serial additive MAP algorithm for both block log-BCJR and SW-log-BCJR. Extension to a parallel implementation is straightforward. Figure 37 shows the implementation of Equation (50) for the forward recursion using a lookup table for evaluation of $log(1+e^{-x})$, and subtraction of $max_j\{A_k(S_j)\}$ from $A_k(S_j)$ is used for normalization to prevent buffer overflow. The circuit for maximization can be implemented simply by using a comparator and selector with feedback operation. Figure 38 shows the implementation of Equation (51) for the backward recursion, which is similar to Figure 37. A circuit for computation of $log(P_k(u|y))$ from Equation (36) using Equation (52) for final computation of bit reliability is shown in Figure 39. In Figure 39, switch 1 is in position 1 and switch 2 is open at the start of operation. The circuit accepts $\sum_k(S_i,u)$ for $i = 1$, then switch 1 moves to position 2 for feedback operation. The circuit performs the operations for $i = 1,2,\cdots,N$. When the circuit accepts $\sum_k(S_i,u)$ for $i = N$, switch 1 goes to position 1 and switch 2 is closed. This operation is done for $u=1$ and $u=0$. The difference between $log(P_k(1|y))$ and $log(P_k(0|y))$ represents the reliability value required for turbo decoding, i.e., the value of L_k in Equation (67).

We propose two simplifications to be used for computation of $log(1+e^{-x})$ without using a lookup table.

Approximation 1: We used the approximation $log(1+e^{-x}) \approx -ax+b$, $0 < x < b/a$ where $b=log(2)$, and we selected $a=0.3$ for the simulation. We observed about 0.1dB degradation compared with the full MAP algorithm for the code. The parameter a should be optimized, and it may not necessarily be the same for the computation of Equation (50), Equation (51), and $log(P_k(u|y))$ from Equation (36) using Equation (52). We call this "linear" approximation.

Approximation 2: We take

$$\log(1+e^{-x}) \approx \begin{cases} 0 & if\ x > \eta \\ c & if\ x < \eta \end{cases}$$

We selected $c=log(2)$ and the threshold $\eta = 1.0$ for our simulation. We observed about 0.2 dB degradation compared with the full MAP algorithm for the code. This threshold should be optimized for a given SNR, and it may not necessarily be the same for the computation of Equation (50), Equation (51), and $log(P_k(u|y))$ from Equation (36) using Equation (52). If we use this approximation, the log-BCJR algorithm can be built based on addition, comparison, and selection operations without requiring a lookup table, which is similar to a Viterbi algorithm implementation. We call this "threshold" approximation. At most, 8 to 10 bit representation suffices for all operations.

1.3.8 Trellis Termination

If needed, the encoder in Figure 28 may generate a $n(N+M), N)$ block code, where the M tail bits of encoder 2 and encoder 3 are not transmitted. Since the component encoders are recursive, it is not sufficient to set the last M information bits to zero in order to drive all the encoder to the all zero state, i.e. to terminate the trellis. The termination (tail) sequence depends on the state of each component encoder after N bits, which makes it impossible to terminate the component encoders with just M bits. This issue has not been resolved in previously proposed turbo code implementations. Fortunately, the simple stratagem illustrated in Figure 33 is sufficient to terminate the trellis at the end of the block. (The code shown is not important). Here the switch is in position "A" for the first N clock cycles and is in position "B" for M additional cycles, which will flush the encoders with zeros. The decoder does not assume knowledge of the M tail bits. The same termination method may be used for all encoders.

1.3.9. Weight Distribution

In order to estimate the performance of a code, it is necessary to have information about its minimum distance, weight distribution, or actual code geometry, depending on the accuracy required for the bounds or approximations. The challenge is in finding the pairing of codewords from each individual encoder, induced by a particular set of interleavers. Intuitively, we would like to avoid joining low-weight codewords from one encoder with low-weight words from the other encoders. In the example of Figure 28, the component codes have distances 5, 2 and 2. This will produce a worst-case minimum distance of 9 for the

overall code. Note that this would be if the encoders were not recursive since, in this case, the minimum weight word for all three encoders is generated by the input sequence $u=(00...0000100...000)$ with a single "1", which will appear again in the other encoders, for any choice of interleavers. This motivates the use of recursive encoders, where the key ingredient is the recursiveness and not the fact that the encoders are systematic. For our example, the input sequence $u=(00...00100100...000)$ generates a low weight codeword with weight 6, for the first encoder. If the interleavers do not "break" this input pattern, the resulting code-words weight will be 14. In general weight-2 sequences with $2+3t$ zeros separating the 1's would result in a total weight of $14+6t$ if there were no permutations. permutations before the second and third encoders, a 2 sequence with its 1's separated by $2+3t_1$ zeros will be permuted into two other weight-2 sequences with 1's separated by $2+3t_i$ zeros, $i=2,3,..$ where each t_i is defined as a multiple of 1/3. If any t_i is not an integer, the corresponding encoded output will have a high weight because then the convolutional code output is non-terminating (until the end of the block). If all t_i's are integers, the total encoded weight will be $14+2\sum_{i=1}^{3} t_i$. Thus, one of the considerations in designing the interleaver is to avoid integer triplets (t_1, t_2, t_3) that are simultaneously small in all three components. In fact, it would be nice to design an interleaver to guarantee that the smallest value of $\sum_{i=1}^{3} t_i$ (for integer t_i) grows with the block size N.

For comparison we consider the same encoder structure in Figure 28, except with the roles of ga and gb reversed. Now the minimum distances of the three component codes are 5, 3, and 3, producing an overall minimum distance of 11 for the total code without any permutations. This is apparently a better code, but it turns out to be inferior as a turbo code. This paradox is explained by again considering the critical weight-2 data sequences. For this code, weight-2 sequences with $1+2t_i$ zeros separating the two 1's produce self-terminating output and hence low-weight encoded words. In the turbo encoder, such sequences will be permuted to have separations $1+2t_i$, $i= 2, 3$, for the second and third encoders, where now each t_i is defined as a multiple of 1/2. But now the total encoded weight for integer triplets (t_1, t_2, t_3) is $11+\sum_{i=1}^{3} t_i$. Notice how this weight grows only half as fast with $\sum_{i=1}^{3} t_i$ as the previously calculated weight for the original code. If $\sum_{i=1}^{3} t_i$ can be made to grow with block size by proper choice of interleaver, then clearly it is important to choose component codes that cause the overall weight to grow as fast as possible with the individual separations t_i. This consideration outweighs the criterion of selecting component codes that would produce the highest minimum distance if unpermuted.

There are also many weight-n, n = 3, 4, 5, ..., data sequences that produce self-terminating output and hence low encoded weight. However, these sequences are much more likely to be broken up by the random interleavers than the weight-2 sequences and are therefore likely to produce non-terminating output from at least one of the encoders. Thus, turbo code structures, which would have low minimum distances if unpermuted, can still perform well if the low-weight codewords of the component codes are produced by input sequences with weight higher of two.

1.3.10. Weight Distribution with random interleavers

Now we briefly examine the issue of whether one or more random interleavers can avoid matching small separations between the 1's of a weight-2 data sequence with equally small separations between the 1's of its permuted version(s). Consider for example a particular weight-2 data sequence $(...001001000...)$ which corresponds to a low weight codeword in each of the encoders of Figure 28. If we randomly select an interleaver of size N, the probability that this sequence will be permuted into another sequence of the same form is roughly $2/N$ (assuming that N is large, and ignoring minor edge effects). The probability that such an unfortunate pairing happens for at least one possible position of the original sequence (...001001000...) within the block size of N, is approximately $1-(1-2/N)^N \approx 1-e^{-2}$. This implies that the minimum distance of a two-code turbo code constructed with a random permutation is not likely to be much higher than the encoded weight of such an unpermuted weight-2 data sequence, e.g. 14 for the code in Figure 28. (For the worst case permutations, the d_{min} of the code is still 9, but these permutations are highly unlikely if chosen randomly). By contrast, if we use three codes and two different interleavers, the probability that a particular sequence $(...001001000...)$ will be reproduced by both interleavers is only $(2/N)^2$. Now the probability of finding such an unfortunate data sequence somewhere within the block of size N is roughly $1-(1-2/N)^N \approx$

$4/N$. Thus it is probable that a three-code turbo code using two random interleavers will see an increase in its minimum distance beyond the encoded weight of an unpermuted weight-2 data sequence. This argument can be extended to account for other weight-2 data sequences which may also produce low weight codewords, e.g. *(...00100(000)' 1000...)*, for the code in Figure 28. For comparison, let us consider a weight-3 data sequence such as *(...0011100...)* which for our example corresponds to the minimum distance of the code (using no permutations). The probability that this sequence is reproduced with one random interleaver is roughly $6/N^2$, and the probability that some sequence of the form *(...0011100...)* is paired with another of the same form is $1-(1-6/N^2)^N \approx 6/N$. Thus for large block sizes, the bad weight-3 data sequences have a small probability of being matched with bad weight-3 permuted data sequences, even in a two-code system. For a turbo code using q codes and q-1 random interleavers this probability is even smaller, $1-(1-(6/N^2)^{q-1})^N \approx (6/N)(6/N^2)^{q-2}$. This implies that the minimum distance codeword of the turbo code in Figure 28 is more likely to result from a weight-2 data sequence of the form *(...001001000...)* than from the weight-3 sequence *(...0011100...)* that produces the minimum distance in the unpermuted version of the same code. Higher weight sequences have even smaller probability of reproducing themselves after being passed through a random interleaver. For a turbo code using q codes and q-1 interleavers, the probability that a weight-n data sequence will be reproduced somewhere within the block by all q-1 permutations is of the form $1-(1-(\beta/N^{n-1})^{q-1})^N$ where β is a number that depends on the weight-n data sequence but does not increase with block size N. For large N, this probability is proportional to $1/N^{nq-n-q}$, which falls off rapidly with N, when n and q are greater than two. Furthermore, the symmetry of this expression indicates that increasing either the weight of the data sequence n or the number of codes q has roughly the same effect on lowering this probability. In summary, from the above arguments we conclude that weight-2 data sequences are an important factor in the design of the component codes, and that higher weight have decreasing importance. Also, increasing the number of coders may result in better turbo codes.

The minimum distance is not the most important quantity of the turbo code, except for its asymptotic performance, at very high E_b/N_o . At moderate SNRs, the weight distribution for the first several possible weights is necessary to compute the code performance. Estimating the complete weight distribution of these codes for large N and fixed interleavers is still an open problem. However, it is possible to estimate the weight distribution for large N for random interleavers by using probabilistic arguments.

1.3.11. Interleaver design

Interleavers should be capable of spreading low-weight input sequences so that the resulting codeword has high weight. Block interleavers, defined by a matrix with υr rows and υc columns such that $N = \upsilon r \times \upsilon c$, may fail to spread certain sequences. For example, the weight 4 sequence shown in Figure 39 cannot be broken by a block interleaver. In order to break such sequences random interleavers are desirable. Block interleavers are effective if the low-weight sequence is confined to a row. If low-weight sequences (which can be regarded as the combination of lower weight sequences) are confined to several consecutive rows, then the υc columns of the interleaver should be sent in a specified order to spread as much as possible the low-weight sequence. As can be observed in the example in Figure 39, the sequence 1001 will still appear at the input of the encoders for any possible column permutation. Only if we permute the rows of the interleaver in addition to its columns it is possible to break the low-weight sequences. Appropriate selection of a, and q for rows and columns depends on the particular set of codes used and on the specific low-weight sequences that we would like to break. We have also designed random permutations (interleavers) by generating random integers i , $1 \le i \le N$, without replacement. We define a "S-random" permutation as follows: "each randomly selected integer is compared to S previously selected integers. If the current selection is equal to any S previous selections within a distance of $\pm S$, then the current selection is rejected. This process is repeated until all N integers are selected. While the searching time increases with S, we observed that choosing $S < (N/2)^{1/2}$ usually produces a solution in reasonable time. (For $S = 1$ we have a purely random interleaver). In the simulations we used $S = 11$ for $N = 256$ and $S = 31$ for $N = 4096$.

The advantage of using three or more constituent codes is that the corresponding two or more interleavers have a better chance to break sequences that were not taken care by another interleaver. The disadvantage is that, for an overall desired code rate, each code must be punctured more, resulting in weaker constituent codes. It has been shown that randomly selected interleavers and interleavers based on the row-column permutation described above. In general, randomly selected permutations are good for low SNR operation (e.g., PCS applications requiring $P_b(e) = 10^{-3}$) where the overall weight distribution of the code is more important than the minimum distance.

1.3.12. Performance with two codes

The performance obtained by turbo decoding the code with two constituent codes (1, *gb/ga*), where *ga* =(37)*octal* and *gb* =(21)*octal* , and with random permutations of lengths $N = 4096$ and (Note that the components of the $\tilde{L}$'s corresponding to the tail bits are set to zero for all iterations). $N = 16384$ is compared in Figure 40 to the capacity of a binary-input Gaussian channel for rate $r = 1/4$. The best performance curve in Figure 40 is approximately 0.7 dB from the Shannon limit at BER=10^{-4} .

1.3.13. Performance with Unequal Rate Encoders.

We now extend the results to encoders with unequal rates with two $K = 5$ constituent codes (1,*gb/ga* , *gc/ga)* and (*gb/ga*), where *ga* = (37)*octal* , *gb* =(33)*octal* and *gc* =(25)*octal* . This structure improves the performance of the overall, rate 1/4, code, as shown in Figure 40. This improvement is due to the fact that we can avoid using the interleaved information data at the second encoder and that the rate of the first code is lower than that of the second code. For PCS applications, short interleaver should be used, since the vocoder frame is usually 20ms. Therefore 192 and 256 bits interleavers as an example, corresponding to 9.6 and 13 Kbps. The performance of codes with short interleaver is shown in Figure 41 for the $K= 5$ codes described above for random permutation and row-column permutation with $a = 2$ for rows and $a = 4$ for columns.

1.3.14. Performance with Three Codes.

The performance of a three-code turbo code with random interleavers is showing Figure 42 for N=4096. The three recursive codes shown in Figure 28 where used for $K = 3$. Three recursive codes with *ga* = (13)*octal* and *gb* =(11)*octal* were used for $K = 4$. Using $K = 4$ code has better performance than several others. In Figure 42, the performance of the $K = 4$ code was improved by going to 30 iterations and using a S-random interleaver with S=31. For shorter blocks (192 and 256), the results are shown in Figure 41 where it can be observed that approximately 1 dB SNR is required for BER=10^{-3} , which implies a CDMA capacity $C = 0.8\eta$ We have noticed that the slope of the BER curve changes around BER=10^{-5} (flattening effect) if the interleaver is not designed properly to maximize d_{min} or is chosen at random.

1.4. Comparison Between Parallel and Serially Multiple Concatenated Codes

In this section, we will use the bit-error probability bounds previously derived to compare the performance of parallel and serially multiple concatenated block and convolutional codes.

1.4.1. Parallel and Serially Multiple Concatenated Block Codes

To obtain a fair comparison, we have chosen the following PMCBC and SMCBC: The PMCBC has parameters (11m, 3m, N) and employs two equal (7,3) systematic cyclic codes with generator $g(D)=(1+D)\ (1 + D + D^3)$; the SMCBC, instead, is a (15m, 4m, N) SMCCC obtained by the concatenation of the (7, 4) Hamming code with a (15, 7) BCH code.

They have almost the same rates ($R_{Cs} = 0.266$ and $R_{Cp} = 0.273$), and have been compared choosing the interleaver length in such a way that the decoding delay due to the interleaver, measured in terms of input information bits, is the same. As an example, to obtain a delay equal to 12 input bits, we must choose an interleaver length $N=12$ for the PMCBC and $N = 12/R^o_C = 21$ for the SMCBC. The results are shown in Figure 25, where we plot the bit-error probability versus the signal-to-noise ratio E_b/N_0 for various input delays. The results show that, for low values of the delay, the performances are almost the same. On the other hand, increasing the delay (and thus the interleaver length N) yields a significant interleaver gain

for the SMCBC and almost no gain for the PMCBC. The difference in performance is 3 dB at $P_b(e) = 10^{-6}$ in favor of the SMCBC.

1.4.2. Parallel and Serially Multiple Concatenated Convolutional Codes

To obtain a fair comparison, we have chosen the following PMCCC and SMCCC: The PMCCC is a rate 1/3 code obtained concatenating two equal rate 1/2, four-state systematic recursive convolutional codes with a generator matrix as in the first row of Table 2. The SMCBC is a rate 1/3 code. It is form using as an outer code the same rate 1/2, four-state code as in the PMCCC and, as an inner code, a rate 2/3, four-state systematic recursive convolutional code with a generator matrix as in the third row of Table 2. Also, in this case, the interleaver lengths have been chosen so as to yield the same decoding delay, due to the interleaver, in terms of input bits. The results are shown in Figure 44, where we plot the bit-error probability versus the signal-to-noise ratio E_b/N_0 for various input delays.

The results show the great difference in the interleaver gain. In particular, the PMCCC shows an interleaver gain going as N^{-1}, whereas the interleaver gain of the SMCCC, as from Expression (31), goes as $N^{-(d^o_f+1)/2} = N^{-3}$, since the free distance of the outer code is equal to 5, which is odd. This means, for $P_b(e) = 10^{-11}$, a gain of more than 2 dB in favor of the SMCCC.

Previous comparisons have shown that serial concatenation is advantageous with respect to parallel concatenation in terms of maximum-likelihood performance. For long interleaver lengths, this significant result remains a theoretical one, as maximum-likelihood decoding is an almost impossible achievement.

2. A SISO MAP Module to Decode Parallel and Serial Multiple Concatenated Codes

Multiple concatenated coding schemes with interleavers comprises a combination of two simple constituent encoders and an interleaver. The parallel concatenation has been shown to yield remarkable coding gains close to theoretical limits, yet admitting a relatively simple iterative decoding technique. The serial concatenation of interleaved codes may offer a superior performance. In both coding schemes, the core of the iterative decoding structure is a soft-input soft-output (SISO) module. Here, we describe the SISO module in a form that continuously updates the maximum a posteriori (MAP) probabilities of input and output code symbols and show how to embed it into iterative decoders for parallel and serially concatenated codes.

2.1. Introduction

Both concatenated coding schemes admit a suboptimum decoding process based on the iterations of the MAP algorithm applied to each constituent code. Here we describe a SISO module that implements the MAP algorithm in its basic form, the extension of it to additive MAP (log-MAP), which is indeed a dual-generalized Viterbi algorithm with correction, and finally extension to the continuous decoding of PMCCC and SMCCC. As examples of applications, we will show the results obtained by decoding two low-rate codes, with very high coding gain.

2.2. Iterative Decoding of Parallel and Serial Concatenated Codes

In this section, we show the block diagram of parallel and serially concatenated codes, together with their iterative decoders, both iterative decoding algorithms need a particular module, named SISO, which implements operations strictly related to the MAP algorithm.

2.2.1. Parallel Concatenated Codes

The block diagram of a PMCCC is shown in Figure 45 (a) (the same construction also applies to block codes). In Figure 45, a rate 1/3 PMCCC is obtained using two rate 1/2 constituent codes (CCs) and an interleaver. For each input information bit, the codeword sent to the channel is formed by the input bit, followed by the parity check bits generated by the two encoders. In Figure 45 (b), the block diagram of the iterative decoder is also shown. It is based on two modules denoted by "SISO," one for each encoder, an interleaver, and a deinterleaver performing the inverse permutation with respect to the interleaver.

The SISO module is a four-port device (quadriport), with two inputs and two outputs. Here, it suffices to say that it accepts as inputs the probability distributions of the information and code symbols labeling the edges of the code trellis, and

forms as outputs an update of these distributions based upon the code constraints. In Figure 45 (b) can be seen that the updated probabilities of the code symbols, are never used by the decoding algorithm.

2.2.2. Serially Multiple Concatenated Codes

The block diagram of a SMCCC is shown in Figure 46 (a) (the same construction also applies to block codes). In Figure 46 (a), a rate 1/3 SMCCC is obtained using as an outer encoder a rate 1/2 encoder, and as an inner encoder a rate 2/3 encoder. An interleaver permutes the output codewords of the outer code before passing them to the inner code. In Figure 46 (b), the block diagram of the iterative decoder is shown. It is based on two modules denoted by "SISO", one for each encoder, an interleaver, and a deinterleaver. The SISO module is the same as described before. In this case, though, both updated probabilities of the input and code symbols are used in the decoding procedure.

2.2.3. Soft-Output algorithms

The SISO module is based on MAP algorithms. These algorithms perform both forward and backward recursions and, thus, require that the whole sequence be received before starting the decoding operations. As a consequence, they can only be used in block-mode decoding. The memory requirement and computational complexity grow linearly with the sequence length.

Some algorithms require only a forward recursion, so that it can be used in continuous-mode decoding. However, its memory and computational complexity grow exponentially with the decoding delay. It is possible to use a MAP symbol-by-symbol decoding algorithm conjugating the positive aspects of other algorithms, i.e., a fixed delay and linear memory and complexity growth with decoding delay. All these algorithms are truly MAP algorithms. To reduce the computational complexity, various forms of suboptimum soft-output algorithms can be used. Two approaches have been taken. The first approach tries to modify the Viterbi algorithm. These augmented outputs include the depth at which all paths are merged, the difference in length between the best and the next-best paths at the point of merging, and a given number of the most likely path sequences. The same concept of augmented output was later generalized for various applications. A different approach to the modification of the Viterbi algorithm comprises of generating a reliability value for each bit of the hard-output signal and is called the soft-output Viterbi algorithm (SOVA). In the binary case, the degradation of SOVA with respect to MAP is small; however, SOVA is not as effective in the nonbinary case. The second approach comprises of revisiting the original symbol MAP decoding algorithms with the aim of simplifying them to a form suitable for implementation.

2.3. The SISO Module

2.3.1. The Encoder

The decoding algorithm underlying the behavior of SISO works for codes admitting a trellis representation. It can be a time-invariant or time-varying trellis, and, thus, the algorithm can be used for both block and convolutional codes

In Figure 47, we show a trellis encoder, characterized by the following quantities (In the following, capital letters *U, C, S, E* will denote random variables and lower-case letters *u, c, s, e* their realizations. The letter *P.[A]* will denote the probability of the event *A*, whereas the letter *P(a)* will denote a function of *a*. The subscript *k* will denote a discrete time, defined on the time index set *K*. Other subscripts, like *i*, will refer to elements of a finite set. Also, "*( )*" will denote a time sequence, whereas "*{ }*" will denote a finite set of elements.):

1. $U = (U_k)_{k \in K}$ is the sequences of input symbols, defined over a time index set K (finite or infinite) and drawn from the alphabet $U = \{ \tilde{u}_1, \ldots, \tilde{u}_{N_I} \}$. To the sequence of input symbols, we associate the sequence of a priori probability distributions: $P.(u;I) = (P_k(u_k ; I))_{k \in K}$ where $P_k(u_k ; I) = P.[U_k = u_k]$.
2. $C = (C_k)_{k \in K}$ is the sequences of output, or code, symbols, defined over the same time index set K, and drawn from the alphabet $C = \{ \tilde{c}_1, \ldots, \tilde{c}_{N_O} \}$. To the sequence of output symbols, we associate the sequence of a priori probability

distributions: $P.(c;I) = \left(P_k(c_k ; I)\right)_{k \in K}$. For simplicity of notation, we drop the dependency of u_k and c_k on k. Thus, $P_k(u_k;I)$ and $P_k(c_k;I)$ will be denoted simply by $P_k(u;I)$ and $P_k(c;I)$ respectively.

2.3.2. The Trellis Section

The dynamics of a time-invariant convolutional code are completely specified by a single trellis section, which describes the transitions (edges) between the states of the trellis at time instants k and $k+1$. A Trellis0 section is characterized by the following:

(1) A set of N states $S == \{ s_1, \ldots, s_n \}$. The state of the trellis at time k is $S_k = s$, with $s \in S$.

(2) A set of $N \times N_I$ edges obtained by the Cartesian product $E = S \times U = \{ e_1, \ldots, e_{NxN_I} \}$ which represents all possible transitions between the trellis states.

The following functions are associated with each edge $e \in E$ (see Figure 48):

(1) The starting state $s^S(e)$ (the projection of e onto S).

(2) The ending state $s^E(e)$.

(3) The input symbol $u(e)$ (the projection of e onto U).

(4) The output symbol $c(e)$.

The relationship between these functions depends on the particular encoder. As an example, in the case of systematic encoders, $(s^E(e)\, c(e))$ also identifies the edge since $u(e)$ is uniquely determined by $c(e)$. In the following, we only assume that the pair $(s^S(e), u(e))$ uniquely identifies the ending state $s^E(e)$; this assumption is always verified, as it is equivalent to say that, given the initial trellis state, there is a one-to-one correspondence between input sequences and state sequences, a property required for the code to be uniquely decodable.

2.3.3. The SISO Algorithm

The SISO module is a four-port device that accepts at the input the sequences of probability distributions $P.(c;I)$ $P.(u;I)$ and outputs the sequences of probability distributions $P.(c;O)$ $P.(u;O)$ based on its inputs and on its knowledge of the trellis section (or code in general). We assume first that the time index set K is finite, i.e., $K = \{1,\ldots,n\}$. The algorithm by which the SISO operates in evaluating the output distributions will be explained in two steps. In the first step, we consider the following algorithm:

(1) At time k, the output probability distributions are computed as

$$\tilde{P}_k(c;O) = \tilde{H}_c \sum_{e:c(e)=c} A_{k-1}[s^S(e)]\; P_k[u(e);I]\; P_k[c(e);I]\; B_k[s^E(e)] \qquad (78)$$

$$\tilde{P}_k(u;O) = \tilde{H}_u \sum_{e:c(e)=u} A_{k-1}[s^S(e)]\; P_k[u(e);I]\; P_k[c(e);I]\; B_k[s^E(e)] \qquad (79)$$

(2) The quantities $A_k(\cdot)$ and $B_k(\cdot)$ are obtained through the forward and backward recursions, respectively, as

$$A_k(s) = \sum_{e:s^E(e)=s} A_{k-1}[s^S(e)]\; P_k[u(e);I]\; P_k[c(e);I] \quad , k = 1,\ldots, n \qquad (80)$$

$$B_k(s) = \sum_{e:s^S(e)=s} B_{k+1}[s^E(e)]\; P_{k+1}[u(e);I]\; P_{k+1}[c(e);I] \quad , k = n-1,\ldots, 0 \qquad (81)$$

with initial values

$$A_0(s) = \begin{cases} 1 & s = S_0 \\ 0 & otherwise \end{cases} \qquad (82)$$

$$B_n(s) = \begin{cases} 1 & s = S_n \\ 0 & otherwise \end{cases} \qquad (83)$$

The quantities $\tilde{H}_c$, $\tilde{H}_u$ are normalization constants defined as follows:

$$\tilde{H}_c \rightarrow \sum_c \tilde{P}_k(c;O) = 1$$

$$\tilde{H}_u \rightarrow \sum_u \tilde{P}_k(u;O) = 1$$

In the second step, from Equations (78) and (79), it is apparent that the quantities $P_k[c(e);I]$ in the first equation and $P_k[u(e);I]$ in the second do not depend on e, by definition of the summation indices, and thus can be extracted from the summations. Thus, defining the new quantities

$$P_k(c;O) = H_c \frac{\tilde{P}_k(c;O)}{P_k(c;I)}$$

$$P_k(u;O) = H_u \frac{\tilde{P}_k(u;O)}{P_k(u;I)}$$

where H_c and H_u are normalization constants such that

$$H_c \rightarrow \sum_c P_k(c;O) = 1$$

$$H_u \rightarrow \sum_u P_k(u;O) = 1$$

It can be easily verified that they can be obtained through the expressions

$$P_k(c;O) = H_c \tilde{H}_c \sum_{e:c(e)=c} A_{k-1}[s^S(e)] P_k[u(e);I] B_k[s^E(e)] \quad (84)$$

$$P_k(u;O) = H_u \tilde{H}_u \sum_{e:c(e)=u} A_{k-1}[s^S(e)] P_k[c(e);I] B_k[s^E(e)] \quad (85)$$

where the A's and B's satisfy the same recursions previously introduced in Equation (80).

The new probability distributions $P_k(u;O)$ and $P_k(c;O)$ represent a smoothed version of the input distributions $P_k(c;I)$ and $P_k(u;I)$, based on the code constraints and obtained using the probability distributions of all symbols of the sequence but the kth ones, $P_k(c;I)$ and $P_k(u;I)$. In the literature of PMCCC decoding, $P_k(u;O)$ and $P_k(c;O)$ would be called extrinsic information. They represent the added value of the SISO module to the a priori distributions $P_k(u;I)$ and $P_k(c;I)$. Basing the SISO algorithm on $P_k(\cdot;O)$ instead of on $\tilde{P}_k(.;O)$ simplifies the block diagrams, and related software and hardware, of the iterative schemes for decoding concatenated codes. The SISO module is then represented as in Figure 49.

Previously proposed algorithms were not in a form suitable for working with a general trellis code. Most of them assumed binary input symbols, some also assumed systematic codes, and none (not even the original BCJR algorithm) could cope with a trellis having parallel edges. As can be noticed, (from all summations involved in the equations that define the SISO algorithm) we work on trellis edges rather than on pairs of states. . This makes the algorithm completely general and capable of coping with parallel edges and also with encoders with rates greater than one, like those encountered in some concatenated schemes.

2.3.4. Computation of Input and Output Bit Extrinsic Information

In this subsection, bit extrinsic information is derived from the symbol extrinsic information using Equations (84) and (85). Consider a rate k_o/n_o trellis encoder such that each input symbol U comprises of k_o bits and each output symbol C comprises of n_o bits. Assume

$$P_k(c;I) = \prod_{j=1}^{n_o} P_{k,j}(c^j;I) \quad (86)$$

$$P_k(u;I) = \prod_{j=1}^{k_o} P_{k,j}(u_j;I) \quad (87)$$

where $c^j \in \{0,1\}$ denotes the value of the jth bit $C^j{}_k$ of the output symbol $C_k = c; j = 1,\ldots,n_o$, and $u^j \in \{0,1\}$ denotes the value of the jth bit $U^j{}_k$ of the output symbol $U_k = u; j = 1,\ldots,k_o$, . This assumption is valid in an iterative decoding when bit

interleavers rather than symbol interleavers are used. One should be cautious when using $P_k(c; I)$ as a product for those encoders in a concatenated system where the output C in Figure 47 is connected to a channel. For such cases, if, for an example, a nonbinary input additive white Gaussian noise (AWGN) channel is used, this assumption usually is not needed (this will be discussed shortly), and $P_k(c; I)=P_k(c|y)=P_k(y|x(c))P(c)/P(y)$, where y is the complex received sample(s) and $x(c)$ is the transmitted nonbinary symbol(s). Then, for binary input memoryless channels, $P_k(y|x(c))$ can be written as a product. After obtaining symbol probability distributions $P_k(c; O)$ and $P_k(u; O)$ from Equations (84) and (85) by using Equations (79) and (81), it is easy then to show that the input and output bit extrinsic information can be obtained as

$$P_{k,j}(c^j;O) = H_{c^j} \sum_{c:C_j^i=c^j} P_k(c;O) \prod_{i=1,i\neq j}^{n_0} P_{k,j}(c^i;I) \qquad (88)$$

$$P_{k,j}(u^j;O) = H_{u^j} \sum_{u:U_j^i=u^j} P_k(u;O) \prod_{i=1,i\neq j}^{n_0} P_{k,j}(u^i;I) \qquad (89)$$

where H_{c^j} and H_{u^j} are normalization constants such that

$$\tilde{H}_{c^j} \rightarrow \sum_{c^j \in \{0,1\}} P_{k,j}(c^j;O) = 1$$

$$\tilde{H}_{u^j} \rightarrow \sum_{u^j \in \{0,1\}} P_{k,j}(u^j;O) = 1$$

Equation (86) is not used for those encoders in a concatenated coded system connected to a channel. To keep the expressions general, as is seen from Equations (80), (81), and (89), $P_k[c(e);I]$ is not represented as a product.

In the following sections, for simplicity of notation, the probability distribution of symbols rather than of bits is considered. The extension of the results to probability distributions of bits based on the above derivations is straightforward.

2.4. The Sliding-Window Soft-Input Soft-Output Module (SW-SISO)

As previous description should have made clear, the SISO algorithm requires that the whole sequence has been received before starting the smoothing process. The reason is due to the backward recursion that starts from the (supposed-known) final trellis state. As a consequence, its practical application is limited to the case when the duration of the transmission is short (n small) or, for n long, when the received sequence can be segmented into independent consecutive blocks, like for block codes or convolutional codes with trellis termination. It cannot be used for continuous decoding of convolutional codes. This constraint leads to a frame, that rigidity imposed on the system and also reduces the overall code rate.

A more flexible decoding strategy is offered by modifying the algorithm. This modification is in such a way, that the SISO module operates on a fixed memory span and outputs the smoothed probability distributions after a given delay D. This new algorithm is called the sliding-window soft-input soft-output (SW-SISO) algorithm (and module). We propose two versions of the SW-SISO that differ in the way they overcome the problem of initializing the backward recursion without waiting for the entire sequence. From now on, we assume that the time index set K is semi-infinite, i.e., $K=\{1,\ldots,\infty\}$, and that the initial state s_0 is known.

2.4.1. First Version of the Sliding-Window SISO Algorithm (SW-SISO1)

The SW-SISO1 algorithm comprises of the following steps:

(1) Initialize A_0 according to Equation (82).

(2) Forward recursion at time k: Compute the A_k through the forward recursion of Equation (80).

(3) Initialization of the backward recursion (time $k > D$):

$$B_k^{(0)}(s) = A_k(s) \quad \forall s \qquad (90)$$

(4) Backward recursion: It is performed according to Equation (81) from iterations $i=1$ to $i=D$ as:

$$B_k^{(i)}(s) = \sum_{e:s^S(e)=s} B_{k-i+1}^{(i-1)}[s^E(e)]\ P_k[u(e);I]\ P_k[c(e);I] \qquad (91)$$

and

$$B_{k-D}(s) = B_{k-D}^{(D)}(s) \quad \forall s \quad (92)$$

(5) The probability distributions at time k -D are computed as

$$P_{k-D}(c;O) = H_c\ \tilde{H}_c \sum_{e:c(e)=c} A_{k-D-1}[s^S(e)]\ P_{k-D}[u(e);I]\ B_{k-D}[s^E(e)] \qquad (93)$$

$$P_{k-D}(u;O) = H_c\ \tilde{H}_c \sum_{e:u(e)=u} A_{k-D-1}[s^S(e)]\ P_{k-D}[c(e);I]\ B_{k-D}[s^E(e)] \qquad (94)$$

2.4.2. The Second Simplified Version of the Sliding-Window SISO Algorithm (SW-SISO2)

A further simplification of the sliding-window SISO algorithm, which is similar to SW-SISO1 except for the backward initial condition, that significantly reduces the memory requirements comprises of the following steps:

(1) Initialize A_0 according to Equation (82).

(2) Forward recursion at time k, $k > D$: Compute the A_{k-D} through the forward recursion

$$A_{k-D}(s) = \sum_{e:s(e)=s} A_{k-D-1}[s^S(e)]\ P_{k-D}[u(e);I]\ P_{k-D}[c(e),I]\ , \quad k > D \qquad (95)$$

(3) Initialization of the backward recursion (time $k > D$):

$$B_k^{(0)}(s) = \frac{1}{N} \quad \forall s \qquad (96)$$

(4) Backward recursion (time $k > D$): It is performed according to Equation (91) as before.

(5) The probability distributions at time $k - D$ are computed according to Equations (93) and (94) as before.

2.4.3. Memory and Computational Complexity

2.4.3.1.- Algorithm SW-SISO1.

For a convolutional code with parameters (k_0, n_0) and number of states N, so that $N_I = 2^{k_0}$ and $N_O = 2^{n_0}$, the algorithm SW-SISO1 requires storage of $N \times D$ values of A's and $D(N_I + N_O)$ values of the input unconstrained probabilities $P_k(u; I)$ and $P_k(c; I)$. Moreover, to update the A's and B's for each time instant, it needs to perform $2 \times N \times N_I$ multiplications and N additions of N_I numbers. To output the set of probability distributions at each time instant, we need a D-times long backward recursion. Thus, overall the computational complexity requires the following: $2(D+1)\times N\times N_I$ *multiplications and* $(D+1)\times N\times(N_I - 1)$ *additions.*

2.4.3.2. Algorithm SW-SISO2.

This simplified version of the sliding-window SISO algorithm does not require the storage of the $N\times D$ values of A's, as they are updated with a delay of D steps. As a consequence, only N values of A's and $D(N_I + N_O)$ values of the input unconstrained probabilities $P_k(u; I)$ and $P_k(c; I)$ need to be stored. The computational complexity is the same as that for the previous version of the algorithm. However, since the initialization of the B recursion is less accurate, a larger value of D may be necessary.

2.5. The Additive SISO Algorithm (A-SISO)

The sliding-window SISO algorithms solve the problems of continuously updating the probability distributions, without requiring trellis terminations. Their computational complexity, however, is still high when compared to other suboptimal algorithms like SOVA. This is due mainly to the fact that they are multiplicative algorithms. We overcome this drawback by proposing the additive version of the SISO algorithm. Clearly, the same procedure can be applied to its two sliding-window versions, SW-SISO1 and SW-SISO2.

To convert the previous SISO algorithm from multiplicative to additive form, we exploit the monotonicity of the logarithm function, and use for the quantities $P(u;\cdot),P(c;\cdot)$, A, and B their natural logarithms, according to the following definitions:

$$\pi_k(c;I) = \log [P_k(c;I)]$$

$$\pi_k(u;I) = \log [P_k(u;I)]$$

$$\pi_k(c;O) = \log [P_k(c;O)]$$

$$\pi_k(u;O) = \log [P_k(u;O)]$$

$$\alpha_k(s) = \log [A_k(s)]$$

$$\beta_k(s) = \log [B_k(s)]$$

With these definitions, the SISO algorithm defined by Equations (84) and (85) and Equations (80) and (81) becomes the following: At time k, the output probability distributions are computed as

$$\pi_k(c;O) = \log \left[\sum_{e:c(e)=c} e^{\left(\alpha_{k-1}[s^S(e)] + \pi_k[u(e);I] + \beta_k[s^E(e)]\right)} \right] + h_c \quad (97)$$

$$\pi_k(u;O) = \log \left[\sum_{e:u(e)=u} e^{\left(\alpha_{k-1}[s^S(e)] + \pi_k[c(e);I] + \beta_k[s^E(e)]\right)} \right] + h_u \quad (98)$$

where the quantities $\alpha_k(\cdot)$ and $\beta_k(\cdot)$ are obtained through the forward and backward recursions, respectively, as

$$\alpha_k(s) = \log \left[\sum_{e:s^E(e)=s} e^{\left(\alpha_{k-1}[s^S(e)] + \pi_k[u(e);I] + \pi_k[c(e),I]\right)} \right] \quad k = 1,\ldots,n \quad (99)$$

$$\beta_k(s) = \log \left[\sum_{e:s^S(e)=s} e^{\left(\beta_{k+1}[s^E(e)] + \pi_{k+1}[u;I] + \pi_{k+1}[c(e),I]\right)} \right] \quad k = n-1,\ldots,0 \quad (100)$$

with initial values

$$\alpha_0(s) = \left\{ \begin{array}{ll} 0 & s = S_0 \\ -\infty & otherwise \end{array} \right\}$$

$$\beta_n(S_i) = \left\{ \begin{array}{ll} 0 & s = S_n \\ -\infty & otherwise \end{array} \right\}$$

The quantities h_c and h_u are normalization constants needed to prevent excessive growth of the numerical values of the α's and β's.

The problem in the previous recursions comprises in the evaluation of the logarithm of a sum of exponential like (in general):

$$a = \log \left[\sum_i^L e^{a_i} \right] \quad (101)$$

To evaluate a in Equation (101), we can use two approximations, with increasing accuracy (and complexity).

The first approximation is

$$a = \log \left[\sum_i^L e^{a_i} \right] \approx a_M \quad (102)$$

where we have defined

$$\alpha_M = \max_i \{ a_i \}, \quad i = 1,\ldots,L$$

This approximation assumes that

$$\alpha_M >> a_i \quad , \quad \forall i = 1, \ldots, L$$

It is almost optimal for medium-high signal-to-noise ratios and leads to performance degradations of the order of 0.5 to 0.7 dB for very low signal-to-noise ratios.

Using Equation (102), the recursions of Equations (99) and (100) become

$$\alpha_k(s) = \max_{e:s^E(e)=s} \left\{ \alpha_{k-1}\left[s^S(e)\right] + \pi_k\left[u(e);1\right] + \pi_k\left[c(e);1\right] \right\} \quad , \quad k = 1, \ldots, n \quad (103)$$

$$\beta_k(s) = \max_{e:s^S(e)=s} \left\{ \beta_{k+1}\left[s^E(e)\right] + \pi_{k+1}\left[u(e);1\right] + \pi_{k+1}\left[c(e);1\right] \right\} \quad , \quad k = n-1, \ldots, 0 \quad (104)$$

and the π's of Equations (97) and (98) become

$$\pi_k(c;O) = \max_{e:c(e)=c} \left\{ \alpha_{k-1}\left[s^S(e)\right] + \pi_k\left[u(e);1\right] + \beta_k\left[s^E(e)\right] \right\} + h_c \quad (105)$$

$$\pi_k(u;O) = \max_{e:u(e)=u} \left\{ \alpha_{k-1}\left[s^S(e)\right] + \pi_k\left[c(e);1\right] + \beta_k\left[s^E(e)\right] \right\} + h_u \quad (106)$$

When the accuracy of the previously proposed approximation is not sufficient, we can evaluate "a" in Equation (101) using the following recursive algorithm:

$$a^{(1)} = a_1$$

$$a^{(l)} = \max\left\{ a^{(l-1)}, a_l \right\} + \log\left[1 + e^{-\left| a^{(l-1)} - a_l \right|} \right] \quad , \quad l = 2, \ldots, L$$

$$a = a^{(L)}$$

To evaluate a, the algorithm needs to perform $(L-1)$ times two kinds of operations: a comparison between two numbers to find the maximum, and the computation of

$$\log\left[1 + e^{(-\Delta)} \right] \Delta \geq 0$$

The second operation can be implemented using a single-entry look-up table up to the desired accuracy. Therefore, a in Equation (101) can be written as $a=a_M+\delta(a_1, a_2, \ldots, a_L)=max_i^* \{a_i\}$. The second term, $\delta(a_1, a_2, \ldots, a_L)$, is called the correction term and can be computed using a look-up table, as discussed above. Now, if desired, max can be replaced by max* in Equations (103) through (106).

Clearly, the additive form of the SISO algorithm can be applied to both versions of the sliding-window SISO algorithms described in the previous section, with straightforward modifications.

2.6. Applications of the ASW-SISO Module

Consider a PMCCC obtained using as constituent codes two equal rate 1/2 systematic, recursive, 16-state convolutional encoders with generating matrix

$$G(D) = \left[1, \frac{1 + D + D^3 + D^4}{1 + D^3 + D^4} \right]$$

The interleaver length is N=16,384. The overall PMCCC forms a very powerful code for possible use in applications requiring reliable operation at very low signal-to-noise ratios.

The performance of the continuous iterative decoding algorithm, applied to the concatenated code, is obtained by simulation, using the ASW-SISO and the look-up table algorithms. It is shown in Figure 50, where we plot the bit-error probability as a function of the number of iterations of the decoding algorithm for various values of the bit signal-to-noise ratio, E_b/N_0. It can be seen that the decoding algorithm converges up to an error probability of 10^{-5}, for signal-to-noise ratios of 0.2 dB with nine iterations. Moreover, convergence is guaranteed also at signal-to-noise ratios as low as 0.05 dB, which is 0.55 dB from the Shannon capacity limit.

As a second example, we construct the serial concatenation of two convolutional codes (SMCCCs) using as an outer code the rate 1/2, 8-state nonrecursive encoder with generating matrix

$$G(D) = [\, 1 + D + D^3,\ 1 + D \,]$$

and, as an inner code, the rate 1/2, 8-state recursive encoder with generating matrix

$$G(D) = \left[1,\ \frac{1 + D + D^3}{1 + D} \right]$$

The resulting SMCCC has rate 1/4. The interleaver length has been chosen to ensure a decoding delay in terms of input information bits equal to 16,384.

The performance of the concatenated code, obtained by simulation as before, is shown in Figure 51, where we plot the bit-error probability as a function of the number of iterations of the decoding algorithm for various values of the bit signal-to-noise ratio, E_b/N_0. It can be seen that the decoding algorithm converges up to an error probability of 10^{-5}, for signal-to-noise ratios of 0.10 dB with nine iterations. Moreover, convergence also is guaranteed at signal-to-noise ratios as low as -0.10 dB, which is 0.71 dB from the capacity limit.

As a third, and final, example, we compare the performance of a PMCCC and an SMCCC with the same rate and complexity. The concatenated code rate is 1/3, the CCs are four-state recursive encoders (rates 1/2 + 1/2 for the PMCCCs and rates 1/2 + 2/3 for the SMCCCs), and the decoding delays in terms of input bits are equal to 16,384. In Figure 52, we report the bit-error probability versus the signal-to-noise ratio for six and nine decoding iterations. As the curves show, the PMCCC outperforms the SMCCC for high values of the bit-error probabilities. Below 10^{-5} (for nine iterations), the SMCCC behaves significantly better and does not present the "floor" behavior typical of PMCCCs. In particular, at 10^{-6}, the SMCCC has an advantage of 0.5 dB with nine iterations.

3. ADSL systems

Figures 53, 54, 55 and 56 are models for facilitating accurate and concise DMT signal waveform descriptions. In the Figures 53, 54, 55 and 56 Z_i is DMT sub-carrier i (defined in the frequency domain), and x_n is the n^{th} IDFT output sample (defined in the time domain). The DAC and analog processing block construct the continuous transmit voltage waveform corresponding to the discrete digital input samples. More precise specifications for these analog blocks arise indirectly from the analog transmit signal linearity and power spectral density specifications. The use of Figures 53, 54, 55 and 56 as a transmitter reference model allows all initialization signal waveforms to be described through the sequence of DMT symbols, $\{Z_i\}$, required to produce that signal. Allowable differences in the characteristics of different digital to analog and analog processing blocks will produce somewhat different continuous-time voltage waveforms for the same initialization signal.

3.1. ATU-C transmitter reference models

ATM and STM are application options. ATU-C and ATU-R may be configured for either STM bit sync transport or ATM cell transport.

3.1.1 ATU-C transmitter reference model for STM transport

Figure 53 is a block diagram of an ADSL Transceiver Unit-Central office (ATU-C) transmitter showing the functional blocks and interfaces for the downstream transport of STM data.

The basic STM transport mode is bit serial. The framing mode used determines if byte boundaries, if present at the V-C interface, shall be preserved. Outside the ASx/LSx serial interfaces data bytes are transmitted MSB first. All serial processing in the ADSL frame (e.g., CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. As a result, the first incoming bit (outside world MSB) shall be the first processed bit inside the ADSL (ADSL LSB). ADSL equipment shall support at least bearer channels AS0 and LS0 downstream. Support of other bearer channels is optional. Two paths are shown between the Mux/Sync control and Tone ordering; the "fast" path provides low latency; the interleaved path provides very low error rate and greater latency.

An ADSL system supporting STM, shall be capable of operating in a dual latency mode for the downstream direction, in which user data is allocated to both paths (i.e. fast and interleaved). An ADSL system supporting STM, shall be capable of operating in a single latency mode for both the downstream and upstream directions, in which all user data is allocated to one path (i.e. fast or interleaved). An ADSL system supporting STM transport may be capable of operating in an optional dual latency mode for the upstream, in which user data is allocated to both paths (i.e. fast and interleaved).

3.1.2 ATU-C transmitter reference model for ATM transport

Figure 54 is a block diagram of an ADSL Transceiver Unit-Central office (ATU-C) transmitter showing the functional blocks and interfaces that are referenced in ITU-T G.992.1 Recommendation for the downstream transport of ATM data.

Byte boundaries at the V–C interface shall be preserved in the ADSL data frame; Outside the ASx/LSx serial interfaces data bytes are transmitted MSB first. All serial processing in the ADSL frame (e.g., CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. The first incoming bit (outside world MSB), will be the first processed bit inside the ADSL (ADSL LSB). The CLP bit of the ATM cell header will be carried in the MSB of the ADSL frame byte (i.e., processed last); ADSL equipment shall support at least bearer channel AS0 downstream). Two paths are shown between the Mux/Sync control and Tone ordering; the "fast" path provides low latency; the interleaved path provides very low error rate and greater latency. An ADSL system supporting ATM transport shall be capable of operating in a single latency mode, in which all user data is allocated to one path (i.e. fast or interleaved). An ADSL system supporting ATM transport may be capable of operating in an optional dual latency mode, in which user data is allocated to both paths (i.e. fast and interleaved).

3.2. ATU-R transmitter reference models

ATM and STM are application options. ATU-C and ATU-R may be configured for either STM bit sync transport or ATM cell transport.

3.2.1 ATU-R transmitter reference model for STM transport

Figure 55 show a block diagram of an ATU-R transmitter showing the functional blocks and interfaces that are referenced in this Recommendation for the upstream transport of STM.

The basic STM transport mode is bit serial. The framing mode used determines if byte boundaries, if present at the V-C interface, shall be preserved. Outside the LSx serial interfaces data bytes are MSB transmitted first. All serial processing in the ADSL frame (e.g., CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. As a result, the first incoming bit (outside world MSB) will be the first processed bit inside the ADSL (ADSL LSB). ADSL equipment shall support at least bearer channel LS0 upstream. Two paths are shown between the Mux/Sync control and Tone ordering; the "fast" path provides low latency; the interleaved path provides very low error rate and greater latency. An ADSL system supporting STM shall be capable of operating in a dual latency mode for the downstream direction, in which user data is allocated to both paths (i.e. fast and interleaved). An ADSL system supporting STM shall be capable of operating in a single latency mode for both the downstream and upstream directions, in which all user data is allocated to one path (i.e. fast or interleaved). An ADSL system supporting STM transport may be capable of operating in an optional dual latency mode for the upstream, in which user data is allocated to both paths (i.e. fast and interleaved).

3.2.2 ATU-R transmitter reference model for ATM transport

Figure 56 show a block diagram of an ATU–R transmitter showing the functional blocks and interfaces that are referenced in this Recommendation for the upstream transport of ATM data.

Byte boundaries at the T–R interface shall be preserved in the ADSL data frame. Outside the LSx serial interfaces data bytes are transmitted MSB first in accordance with ITU-T Recommendations I.361 and I.432. All serial processing in the ADSL frame (e.g. CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. As a result, the first incoming bit (outside world MSB) will be the first processed bit inside the ADSL

(ADSL LSB), and the CLP bit of the ATM cell header will be carried in the MSB of the ADSL frame byte (i.e., processed last). ADSL equipment shall support at least bearer channel LS0 upstream. Two paths are shown between the Mux/Sync control and Tone ordering; the "fast" path provides low latency, the interleaved path provides very low error rate and greater latency. An ADSL system supporting ATM transport shall be capable of operating in a single latency mode, in which all user data is allocated to one path (i.e. fast or interleaved). An ADSL system supporting ATM transport may be capable of operating in an optional dual latency mode, in which user data is allocated to both paths (i.e. fast and interleaved).

3.3 Transport Capacity.

An ADSL system may transport up to seven user data streams on seven bearer channels simultaneously up to four independent downstream simplex bearers (unidirectional from the network operator (i.e. V-C interface) to the CI (i.e. T-R interface))

An ADSL system may transport up to three duplex bearers (bi-directional between the network operator and the CI).

The three duplex bearers may alternatively be configured as independent unidirectional simplex bearers, and the rates of the bearers in the two directions (network operator toward CI and vice versa) do not need to match.

All bearer channel data rates shall be programmable in any combination of integer multiples of 32 kbit/s.

The ADSL data multiplexing format is flexible enough to allow other transport data rates, such as channelizations based on existing 1.544 Mbit/s, but the support of these data rates (non-integer multiples of 32 kbit/s) will be limited by the ADSL system's available capacity for synchronization.

The maximum net data rate transport capacity of an ADSL system will depend on the characteristics of the loop on which the system is deployed, and on certain configurable options that affect overhead. The ADSL bearer channel rates shall be configured during the initialization and training procedure.

The transport capacity of an ADSL system per se is defined only as that of the bearer channels. When, however, an ADSL system is installed on a line that also carries POTS or ISDN signals the overall capacity is that of POTS or ISDN plus ADSL

A distinction is made between the transport of synchronous (STM) and asynchronous (ATM) data. An ATU-x shall be configured to support STM transmission or ATM transmission. Bearer channels configured to transport STM data can also be configured to carry ATM data. ADSL equipment may be capable of simultaneously supporting both ATM and STM transport.

If an ATU-x supports a particular bearer channel it shall support it through both the fast and interleaved paths.

In addition, an ADSL system may transport a Network Timing Reference (NTR).

3.3.1. Transport of STM data

ADSL systems transporting STM shall support the simplex bearer channel AS0 and the duplex bearer channel LS0 downstream. Bearer channels AS0, LS0, and any other bearer channels supported shall be independently allocable to a particular latency path as selected by the ATU-C at start-up. The system shall support dual-latency downstream.

ADSL systems transporting STM shall support the duplex bearer channel LS0 upstream using a single latency path.

Bearer channel AS0 shall support the transport of data at all integer multiples of 32 kbit/s from 32 kbit/s to 6144 kbit/s. Bearer channel LS0 shall support 16 kbit/s and all integer multiples of 32 kbit/s from 32 kbit/s to 640 kbit/s.

When AS1, AS2, AS3, LS1 and LS2 are provided, they shall support the range of integer multiples of 32 kbit/s shown in Table 4. Support for data rates based on non-integer multiples of 32 kbit/s is also optional.

Table 4 shows the required 32 kbit/s integer multiples for transport of STM.

Table 4. Required 32 kbit/s integer multiples for transport of STM

6Bearer channel	Lowest Required Integer Multiple	Largest Required Integer Multiple	Corresponding Highest Required Data Rate (kbit/s)
AS0	1	192	6144
AS1	1	144	4608
AS2	1	96	3072
AS3	1	48	1536
LS0	1	20	640
LS1	1	20	640
LS2	1	20	640

Table 5 illustrates the data rate terminology and definitions used for STM transport.

Table 5. Data Rate Terminology for STM transport

				Data Rate	Equation (kbits/s)	Reference Point
		STMdata rate	=	"Net data rate"	$\Sigma(B_I, B_F)$ X 32 (NOTE)	ASx + LSx
"Net data rate"	+	Frame overhead rate	=	"Aggregate data rate"	$\Sigma(K_I, K_F)$ X 32	A
"Aggregate data rate"	+	RS Coding overhead rate	=	"Total data rate"	$\Sigma(N_I, N_F)$ X 32	B
"Total data rate"	+	Trellis Coding overhead rate	=	Line rate	Σb_i X 4	U
NOTE – Net data rate increase by 16 kbit/s if a 16 kbit/s "C"-channel is used.						

3.3.2. Transport of ATM data

An ADSL system transporting ATM shall support the single latency mode at all integer multiples of 32kbit/s up to 6.144 Mbit/s downstream and up to 640 kbit/s upstream. For single latency, ATM data shall be mapped to bearer channel AS0 in the downstream direction and to bearer channel LS0 in the upstream direction.

The need for dual latency for ATM services depends on the service/application profile, and is under study by the ITU. One of three different "latency classes" may be used: Single latency, not necessarily the same for each direction of transmission, Dual latency downstream, single latency upstream, Dual latency both upstream and downstream.

ADSL systems transporting ATM shall support bearer channel AS0 downstream and bearer channel LS0 upstream, with each of these bearer channels independently allocable to a particular latency path as selected by the ATU-C at start-up. Therefore, support of dual latency is optional for both downstream and upstream.

If downstream ATM data are transmitted through a single latency path (i.e., 'fast' only or 'interleaved' only), only bearer channel AS0 shall be used. and it shall be allocated to the appropriate latency path. If downstream ATM data are transmitted through both latency paths (i.e., 'fast' and 'interleaved'), only bearer channels AS0 and AS1 shall be used, and they shall be allocated to different latency paths.

Similarly, if upstream ATM data are transmitted through a single latency path (i.e., 'fast' only or 'interleaved' only), only bearer channel LS0 shall be used and it shall be allocated to the appropriate latency path. The choice of the fast or interleaved path may be made independently of the choice for the downstream data. If upstream ATM data are transmitted through both

latency paths (i.e., 'fast' and 'interleaved'), only bearer channels LS0 and LS1 shall be used and they shall be allocated to different latency paths.

Bearer channel AS0 shall support the transport of data at all integer multiples of 32 kbit/s from 32 kbit/s to 6144 kbit/s. Bearer channel LS0 shall support all integer multiples of 32 kbit/s from 32 kbit/s to 640 kbit/s. Support for data rates based on non-integer multiples of 32 kbit/s is also optional.

When AS1 and LS1 are provided, they shall support the range of integer multiples of 32 kbit/s shown in Table 4. Data rates based on non-integer multiples of 32 kbit/s is optional.

Bearer channels AS2, AS3 and LS2 shall not be provided for an ATM based ATU-x.

Table 6 illustrates the data rate terminology and definitions used for ATM transport.

Table 6- Data Rate Terminology for ATM transport

				Data Rate	Equation kbits/s)	Ref Point
53 x 8 x ATM cell rate			=	"Net data rate"	$\Sigma(B_I, B_F) \times 32$	ASx + LSx
"Net data rate"	+	Frame overhead rate	=	"Aggregate data rate"	$\Sigma(K_I, K_F) \times 32$	A
"Aggregate data rate"	+	RS Coding overhead rate	=	"Total data rate"	$\Sigma(N_I, N_F) \times 32$	B
"Total data rate"	+	Trellis Coding overhead rate	=	Line rate	$\Sigma b_i \times 4$	U

3.3.3. ADSL system overheads and total bit rates

The total bit rate transmitted by the ADSL system when operating in an optional reduced-overhead framing mode shall include capacity for the data rate transmitted in the ADSL bearer channels and ADSL system overhead (which includes: an ADSL embedded operations channel, EOC; an ADSL overhead control channel, AOC; CRC check bytes; fixed indicator bits for OAM; FEC redundancy bytes). When operating in the full-overhead mode the total bit rate shall also include capacity for the synchronization control bytes and capacity for bearer channel synchronization control.

The internal overhead channels and their rates are shown in Table 7.

Table 7. Internal overhead channel functions and rates

	Downstream rate (kbit/s) minimum / maximum		Upstream rate (kbit/s) minimum / maximum	
	Number of ASx bearer channels > 1	Number of ASx bearer channels = 1	Number of LSx bearer channels > 1	Number of LSx bearer channels = 1
Synchronization control, CRC and AOC; interleaved buffer	32/32	32/32	32/32	32/32
Synchronization control, CRC, EOC and indicator bits; fast buffer	32/32	32/32	32/32	32/32
Total for reduced overhead framing	32 / 64 (NOTE 2)	32 / 64 (NOTE 2)	32 / 64 (NOTE 2)	32 / 64 (NOTE 2)
Synchronization capacity (shared among all bearer channels)	64 / 128 (NOTE 3)	64 / 96 (NOTE 3)	32 / 64 (NOTE 3)	32 / 32 (NOTE 3)
Total (NOTE 1)	128 / 192	128 / 160	96 / 128	96 / 96

NOTE 1 - The overhead required for FEC is not shown in this table.

NOTE 2 - With the reduced overhead framing modes, a 32 kbit/s ADSL system overhead is present in each buffer type. However, when all ASx and LSx are allocated to one buffer type, synchronization control, CRC, EOC, AOC and indicator bits may be carried in a single 32 kbit/s ADSL system overhead present in the buffer type used. With full overhead framing, a 32 kbit/s ADSL system overhead is always present in each buffer type.

NOTE 3 - The shared synchronization capacity includes 32 kbit/s shared among LSx within the interleave buffer, 32 kbit/s shared among LSx within the fast buffer, 32 kbit/s shared among ASx within the interleave buffer, and 32 kbit/s shared among ASx within the fast buffer. The maximum rate occurs when at least one ASx is allocated to each type of buffer; the minimum rate occurs when all ASx and LSx are allocated to one buffer type.

3.4. ATU-C Functional Characteristics.

An ATU-C may support STM transmission or ATM transmission or both. Framing modes that shall be supported, depend upon the ATU-C being configured for either STM or ATM transport. If framing mode k is supported, then modes k-1, ..., 0 shall also be supported..

During initialization, the ATU-C and ATU-R shall indicate a framing mode number 0, 1, 2 or 3, which they intend to use. The lowest indicated framing mode shall be used.

Using framing mode 0 ensure than an STM based ATU-x with an external ATM TC will interoperate with an ATM based ATU-x. Additional modes of interoperation are possible depending upon optional features provided in either ATU-x.

An ATU-C may provide a Network Timing Reference (NTR). This operation shall be independent of any clocking that is internal to the ADSL system.

3.4.1. STM Transmission Protocol Specific functionalities

3.4.1.1. ATU-C input and output V interfaces for STM transport

The functional data interfaces at the ATU-C for STM transport are shown in Figure 57. Input interfaces for the high-speed downstream simplex bearer channels are designated AS0 through AS3; input/output interfaces for the duplex bearer channels are designated LS0 through LS2. There shall also be a duplex interface for operations, administration, maintenance (OAM) and control of the ADSL system.

3.4.1.2. Downstream simplex channels

Four data input interfaces are defined at the ATU-C for the high-speed downstream simplex channels: AS0, AS1, AS2 and AS3 (ASx in general).

3.4.1.3. Downstream/upstream duplex channels

Three input and output data interfaces are defined at the ATU-C for the duplex channels supported by the ADSL system; LS0, LS1, and LS2 (LSx in general). LS0 is also known as the "C" or control channel. It carries the signaling associated with the ASx bearer channels and it may also carry some or all of the signaling associated with the other duplex bearer channels.

3.4.1.4. Payload transfer delay

The one-way transfer delay for payload bits in all bearers (simplex and duplex) from the V reference point at central office end (V-C) to the T reference point at remote end (T-R) for channels assigned to the fast buffer shall be no more than 2 ms. For channels assigned to the interleave buffer it shall be no more than $(4 + (S-1)/4 + SxD/4)$ ms. The same requirement applies in the opposite direction, from the T-R reference point to the V-C reference point.

3.4.1.5. Framing Structure for STM transport

An ATU-C configured for STM transport shall support the full overhead framing structure 0.The support of full overhead framing structure 1 and the reduced overhead framing structures 2 and 3 is optional. Preservation of V-C interface byte boundaries (if present) at the U-C interface may be supported for any of the U-C interface framing structures.An ATU-C configured for STM transport may support insertion of a Network Timing Reference (NTR).

3.4.2. ATM Transmission Protocol Specific functionalities

3.4.2.1. ATU-C input and output V interface for ATM transport

The functional data interfaces at the ATU-C for ATM is shown in Figure 58. The ATM channel ATM0 shall always be provided, the channel ATM1 may be provided for support of dual latency mode. Each channel operates as an interface to a physical layer pipe. When operating in dual latency mode, no fixed allocation between the ATM channels 0 and 1 on one hand and transport of 'fast' and 'interleaved' data on the other hand is assumed. This relationship is configured inside the ATU-C.

Flow control functionality shall be available on the V reference point to allow the ATU-C (i.e. the physical layer) to control the cell flow to and from the ATM layer. This functionality is represented by Tx_Cell_Handshake and Rx_Cell_Handshake. A cell may be transferred from ATM to PHY layer only after the ATU-C has activated the Tx_Cell_Handshake. Similarly a cell may be transferred from the PHY layer to the ATM layer only after the Rx_Cell_Handshake. This functionality is important to avoid cell overflow or underflow in the ATU-C and ATM layers.

There shall also be a duplex interface for operations, administration, maintenance (OAM) and control of the ADSL system.

3.4.2.2 Payload transfer delay

The one-way transfer delay (excluding cell specific functionalities) for payload bits in all bearers (simplex and duplex) from the V reference point at central office end (V-C) to the T reference point at remote end (T-R) for channels assigned to the fast buffer shall be no more than 2 ms.

For channels assigned to the interleave buffer it shall be no more than $(4 + (S-1)/4 + SxD/4)$ ms. The same requirement applies in the opposite direction, from the T-R reference point to the V-C reference point.

3.4.2.3. ATM Cell specific functionalities

3.4.2.3.1. Idle Cell Insertion

Idle cells shall be inserted in the transmitter direction for cell rate de-coupling. Idle cells are identified by the standardized pattern for the cell header given in ITU-T Recommendation I.432.

3.4.2.3.2. Header Error Control (HEC) Generation.

The HEC byte shall be generated in the transmit direction as described in ITU-T Recommendation I.432, including the recommended modulo 2 addition (XOR) of the pattern 01010101_2 to the HEC bits. The generator polynomial coefficient set used and the HEC sequence generation procedure shall be in accordance with ITU-T Recommendation I.432.

3.4.2.3.3. Cell payload scrambling.

Scrambling of the cell payload field shall be used in the transmit direction to improve the security and robustness of the HEC cell delineation mechanism. In addition, it randomizes the data in the information field, for possible improvement of the transmission performance. The self synchronizing scrambler polynomial $X^{43}+1$ and procedures defined in ITU-T Recommendation I.432 shall be implemented.

3.4.2.3.4. Bit timing and ordering

When interfacing ATM data bytes to the AS0 or AS1 bearer channel, the most significant bit (MSB) shall be sent first. The AS0 or AS1 bearer channel data rates shall be integer multiples 32 kbit/s, with bit timing synchronous with the ADSL downstream timing base.

3.4.2.3.5 Cell Delineation.

The cell delineation function permits the identification of cell boundaries in the payload. It uses the HEC field in the cell header. Cell delineation shall be performed using a coding law checking the HEC field in the cell header according to the algorithm described in ITU-T Recommendation I.432. The ATM cell delineation state machine is shown in Figure 59.

In the HUNT state, the delineation process is performed by checking bit by bit for the correct HEC. Once such an agreement is found, it is assumed that one header has been found, and the method enters the PRESYNC state. When byte boundaries are available within the receiving Physical Layer prior to cell delineation as with the framing modes 1, 2 and 3, the cell delineation process may be performed byte by byte. In the PRESYNC state, the delineation process is performed by checking cell by cell for the correct HEC. The process repeats until the correct HEC has been confirmed *DELTA* times consecutively. If an incorrect HEC is found, the process returns to the HUNT state. In the SYNC state the cell delineation will be assumed to be lost if an incorrect HEC is obtained *ALPHA* times consecutively. (With reference to ITU-T Recommendation I.432, no recommendation is made for the values of *ALPHA* and *DELTA* as the choice of these values is not considered to effect interoperability. However, it should be noted that the use of the values suggested in ITU-T Recommendation I.432 (*ALPHA*=7, *DELTA*=6) may be inappropriate due to the particular transmission characteristics of ADSL).

3.4.2.3.6 Header Error Control Verification

The HEC covers the entire cell header. The code used for this function is capable of either: single bit error correction or multiple bit error detection. Error detection shall be implemented as defined in ITU-T Recommendation I.432 with the exception that any HEC error shall be considered as a multiple bit error, and therefore, HEC error correction shall not be performed.

3.4.2.4 Framing Structure for ATM transport

An ATU-C configured for ATM transport shall support the full overhead framing structures 0 and 1. The support of reduced overhead framing structures 2 and 3 is optional. The ATU-C transmitter shall preserve V-C interface byte boundaries (explicitly present or implied by ATM cell boundaries) at the U-C interface, independent of the U-C interface framing structure.

To ensure framing structure 0 interoperability between an ATM ATU-C and an ATM cell TC plus an STM ATU-R (i.e., ATM over STM), transporting ATM cells and not preserving T-R byte boundaries at the U-R interface shall indicate

during initialization that frame structure 0 is the highest frame structure supported. An STM ATU-R transporting ATM cells and preserving T-R byte boundaries at the U-R interface shall indicate during initialization that frame structure 0, 1, 2 or 3 is the highest frame structure supported. An ATM ATU-C receiver operating in framing structure 0 can not assume that the ATU-R transmitter will preserve T-R interface byte boundaries at the U-R interface and shall therefore perform the cell delineation bit-by-bit.

An ATU-C configured for ATM transport may support insertion of a Network Timing Reference (NTR).

3.4.3 Network timing reference (NTR)

3.4.3.1 Need for NTR

Some services require that a reference clock be available in the higher layers of the protocol stack (i.e. above the physical layer); this is used to guarantee end-to-end synchronization of transmit and receive sides. Examples are Voice and Telephony Over ATM (VTOA) and Desktop Video Conferencing (DVC).

To support the distribution of a timing reference over the network, the ADSL system may transport an 8 kHz timing marker as NTR. This 8 kHz timing marker may be used for voice/video playback at the decoder (D/A converter) in DVC and VTOA applications. The 8 kHz timing marker is input to the ATU-C as part of the interface at the V-C reference point.

3.4.3.2 Transport of the NTR

The intention of the NTR transport mechanism is that the ATU-C provides timing information at the U-C reference point to enable the ATU-R to deliver to the T-R reference point timing information that has a timing accuracy corresponding to the accuracy of the clock provided to the V-C reference point. If provided, the NTR shall be inserted in the U-C framing structure as follows:

a) The ATU-C may generate an 8 kHz local timing reference (LTR) by dividing its sampling clock by the appropriate integer (276 if 2.208 MHz is used);

b) It shall transmit the change in phase offset between the input NTR and LTR (measured In cycles of the 2.208 MHz clock, that is units of approximately 452 ns) from the previous superframe to the present one. This shall be encoded into four bits ntr3 - ntr0 (with ntr3 the MSB), representing a signed integer in the –8 to +7 range in 2's-complement notation. The bits ntr3-ntr0 shall be carried in the indicator bits 23 (ntr3) to 20 (ntr0); see Table 9.

c) A positive value of the change of phase offset, D^2f, shall indicate that the LTR is higher In frequency than the NTR.

d) Alternatively, the ATU-C may choose to lock its downstream sampling clock (2.208 MHz) to 276 times the NTR frequency; in that case it shall encode $\Delta^2 f$ to zero.

The NTR, as specified by ANSI Standard T1.101, has a maximum frequency variation of ±32 ppm. The LTR has a maximum frequency variation of ±50 ppm. The maximum mismatch is therefore ±82 ppm. This would result in an average change of phase offset of approximately ± 3.5 clock cycles over one 17 ms superframe, which can be mapped into 4 overhead bits.

One method that the ATU-C may use to measure this change of phase offset is shown in Figure 60.

3.4.4 Framing

This subclause specifies framing of the downstream signal (ATU-C transmitter). Two types of framing are defined: full overhead and reduced overhead. Furthermore, two versions of full overhead and two versions of reduced overhead are defined. The four resulting framing modes are defined in Table 8, and shall be referred to as framing modes 0, 1, 2 and 3.

Table 8- Definition of framing modes

Framing structure	Description
0	Full overhead framing with asynchronous bit-to-modem timing (i.e. enabled synchronization control mechanism)
1	Full overhead framing with synchronous bit-to-modem timing (i.e. disabled synchronization control mechanism)
2	Reduced overhead framing with separate fast and sync byte in fast and interleaved latency buffer respectively (i.e. 64 kbit/s framing overhead)
3	Reduced overhead framing with merged fast and sync byte, using either the fast or the interleaved latency buffer (i.e. 32 kbit/s framing overhead)

Requirements for framing modes to be supported, depend upon the ATU-C being configured for either STM or ATM transport. The ATU-C shall indicate during initialization the highest framing structure number it supports. If the ATU-C indicates it supports framing structure *k*, it shall also support all framing structures *k*-1 to 0. If the ATU-R indicates a lower framing structure number during initialization, the ATU-C shall fall back to the framing structure number indicated by the ATU-R. Outside the ASx/LSx serial interfaces data bytes are transmitted MSB first in accordance with ITU-T Recommendations G.703, G.709, I.361, and I.432. All serial processing in the ADSL frame (e.g., CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. As a result, the first incoming bit (outside world MSB) will be the first processed bit inside the ADSL (ADSL LSB).

3.4.4.1 Data symbols

Figures 53 and 54 show functional block diagrams of the ATU-C transmitter with reference points for data framing. Up to four downstream simplex data channels and up to three duplex data channels shall be synchronized to the 4 kHz ADSL DMT frame rate, and multiplexed into two separate data buffers (fast and interleaved). A cyclic redundancy check (CRC), scrambling, and forward error correction (FEC) coding shall be applied to the contents of each buffer separately, and the data from the interleaved buffer shall then be passed through an interleaving function. The two data streams shall then be tone ordered, and combined into a data symbol that is input to the constellation encoder. After constellation encoding the data shall be modulated to produce an analog signal for transmission across the customer loop.

A bit-level framing pattern shall not be inserted into the data symbols of the frame or superframe structure. DMT frame (i.e. symbol) boundaries are delineated by the cyclic prefix inserted by the modulator. Superframe boundaries are determined by the synchronization symbol and shall also be inserted by the modulator, and which carries no user data.

Because of the addition of FEC redundancy bytes and data interleaving, the data frames (i.e. bit-level data prior to constellation encoding) have different structural appearance at the three reference points through the transmitter. As shown in Figures 53 and 54, the reference points for which data framing will be described in the following subclauses is:

a) **A** (Mux data frame): the multiplexed, synchronized data after the CRC has been inserted

b) **B** (FEC output data frame): the data frame generated at the output of the FEC encoder at the DMT symbol rate, where an FEC block may span more than one DMT symbol period

c) **C** (constellation encoder input data frame): the data frame presented to the constellation coder.

3.4.4.1.1 Superframe structure

ADSL uses the superframe structure shown in Figure 61. Each superframe is composed of 68 data frames, numbered from 0 to 67, which are encoded and modulated into DMT symbols, followed by a synchronization symbol, which carries no user or overhead bit-level data and is inserted by the modulator to establish superframe boundaries. From the bit-level and user data perspective, the DMT symbol rate is 4000 baud (period = 250 μs), but in order to allow for the insertion of the

synchronization symbol the transmitted DMT symbol rate is 69/68*4000 baud. Each data frame within the superframe contains data from the fast buffer and the interleaved buffer. During each ADSL superframe, eight bits shall be reserved for the CRC on the fast data buffer (crc0-crc7), and 24 indicator bits (ib0-ib23) shall be assigned for OAM functions. As shown in Figure 62, the synchronizaton byte of the fast data buffer ("fast byte") carries the CRC check bits in frame 0 and the indicator bits in frames 1, 34, and 35. The fast byte in other frames is assigned in even-/odd-frame pairs to either the EOC or to synchronization control of the bearer channels assigned to the fast buffer.

Bit 0 of the fast byte in an even-numbered frame (other than frames 0 and 34) and bit 0 of the fast byte of the odd-numbered frame immediately following shall be set to "0" to indicate these frames carry a synchronization control information. When they are not required for synchronization control, CRC, or indicator bits, the fast bytes of two successive ADSL frames, beginning with an even-numbered frame, may contain indications of "no synchronization action" or alternatively, they may be used to transmit one EOC message, consisting of 13 bits. The indicator bits are defined in Table 9. Bit 0 of the fast byte in an even-numbered frame (other than frames 0 and 34) and bit 0 of the fast byte of the odd-numbered frame immediately following shall be set to "1" to indicate these frames carry a 13-bit EOC message plus one additional bit, r1. The r1 bit is reserved for future use and shall be set to 1.

Table 9 - Definition of indicator bits, ATU-C transmitter (fast data buffer, downstream direction)

Indicator bit	Definition (see NOTE)
ib0 - ib7	reserved for future use
ib8	FEBE-I
ib9	FECC-I
ib10	FEBE-F
ib11	FECC-F
ib12	LOS
ib13	RDI
ib14	NCD-I (used for ATM only, shall be set to 1 for STM)
ib15	NCD-F (used for ATM only, shall be set to 1 for STM)
ib16	HEC-I (used for ATM only, shall be set to 1 for STM)
ib17	HEC-F (used for ATM only, shall be set to 1 for STM)
ib18- 19	reserved for future use
ib20-23	NTR0 – 3 (if NTR is not transported, ib20-23 shall be set to 1 – they are active low)
NOTE - Because all indicator bits are defined as active low, reserved bits shall be set to 1.	

Eight bits per ADSL superframe shall be used for the CRC on the interleaved data buffer (crc0 - crc7). As shown in Figure 63 and Figure 65, the synchronization byte of the interleaved data buffer ("sync byte") carries the CRC check bits for the previous superframe in frame 0. In all other frames (1 through 67), the sync byte shall be used for synchronization control of the bearer channels assigned to the interleaved data buffer or used to carry an ADSL overhead control (AOC) channel. In the full overhead mode, when any bearer channel appears in the interleave buffer, then the AOC data shall be carried in the LEX byte, and the sync byte shall designate when the LEX byte contains AOC data and when it contains data bytes from the bearer channel. When no bearer channels are allocated in the interleaved data buffer (i.e., all $B_I(ASx) = B_I(LSx) = 0$), the sync byte shall carry the AOC data directly.

3.4.4.1.2 Frame structure (with full overhead)

Each data frame shall be encoded into a DMT symbol. As is shown in Figure 61, each frame is composed of a fast data buffer and an interleaved data buffer, and the frame structure has a different appearance at each of the reference points (A, B, and C). The bytes of the fast data buffer shall be clocked into the constellation encoder first, followed by the bytes of the interleaved data buffer. Bytes are clocked least significant bit first.

Each bearer channel shall be assigned to either the fast or the interleaved buffer during initialization, and a pair of bytes, $[B_F,B_I]$, transmitted for each bearer channel, where B_F and B_I designate the number of bytes allocated to the fast and interleaved buffers, respectively.

The seven $[B_F,B_I]$ pairs to specify the downstream bearer channel rates are: $B_F(ASx)$, $B_I(ASx)$ for X = 0, 1, 2 and 3, for the downstream simplex channels; $B_F(LSx)$, $B_I(LSx)$ for X = 0, 1 and 2, for the (downstream transport of the) duplex channels.

The rules for allocation are as follow:

- For any bearer channel, X, (except the 16 kbit/s C channel option) either $B_F(X)$ = the number of bytes per frame of the fast buffer and $B_I(X) = 0$, *or* $B_F(X) = 0$ and $B_I(X)$ = the number of bytes per frame of the interleaved buffer.
- For the 16 kbit/s C channel option, $B_F(LS0) = 255\ (11111111_2)$ and $B_I(LS0) = 0$, or $B_F(LS0) = 0$ and $B_I(LS0) = 255$.

3.4.4.1.2.1 Fast data buffer (with full overhead)

The frame structure of the fast data buffer shall be as shown in Figure 64, for reference points A and B, which are defined in Figure 53 and 54.

The following shall hold for the parameters shown in Figure 64:

$$C_F(LS0) = 0 \text{ if } B_F(LS0) = 255\ (11111111_2)$$
$$= B_F(LS0) \text{ otherwise}$$
$$N_F = K_F + R_F$$

where R_F = number of FEC redundancy bytes, and

$$K_F = 1 + \sum_{i=0}^{3} B_F(ASi) + A_F + C_F(LS0) + \sum_{j=1}^{2} B_F(LSj) + L_F$$

where

$$A_F = 0 \quad \text{if} \sum_{i=0}^{3} B_F(ASi) = 0 \text{ for } i=0\text{-}3$$
$$= 1 \text{ otherwise}$$

and

$$L_F = 0 \text{ if } B_f(ASi) = 0 \text{ for } i=0\text{-}3 \text{ and } B_f(LSj) = 0 \text{ for } j=0\text{-}2$$
$$= 1 \text{ otherwise (including } B_F(LS0) = 255)$$

At reference point A (Mux data frame) in Figure 53 and Figure 54 the fast buffer shall always contain at least the fast byte. This is followed by $B_F(AS0)$ bytes of channel $AS0$, then $B_F(AS1)$ bytes of channel $AS1$, $B_F(AS2)$ bytes of channel $AS2$ and $B_F(AS3)$ bytes of channel $AS3$. Next come the bytes for any duplex (LSx) channels allocated to the fast buffer. If any $B_F(ASx)$ is non-zero, then both *an AEX* and a *LEX* byte follow the bytes of the last LSx channel, and if any $B_F(LSx)$ is non-zero, the *LEX* byte shall be included.

When $B_F(LS0) = 255$, no bytes are included for the *LS0* channel. Instead, the 16 kbit/s C channel shall be transported in every other *LEX* byte on average, using the sync byte to denote when to add the *LEX* byte to the *LS0* bearer channel.

R_F *FEC* redundancy bytes shall be added to the mux data frame (reference point A) to produce the FEC output data frame (reference point B), where R_F is given in the options used during initialization.

Because the data from the fast buffer is not interleaved, the constellation encoder input data frame (reference point C) is identical to the FEC output data frame (reference point B).

3.4.4.1.2.2. Interleaved data buffer (with full overhead)

The frame structure of the interleaved data buffer is shown in Figure 65 for reference points A and B, which are defined in Figure 53 and 54.

The following shall hold for the parameters shown in Figure 65

$C_I(LS0) = 0$ *if* $B_I(LS0) = 255\ (11111111_2)$

$= B_I(LS0)$ *otherwise*

$N_I = (S \times K_I + R_I) / S$,

where R_I = number of FEC redundancy bytes and S = number of DMT symbols per FEC codeword.

$$K_I = 1 + \sum_{i=0}^{3} B_I(ASi) + A_I + C_I(LS0) + \sum_{j=1}^{2} B_I(LSj) + L_I$$

where

$$A_I = 0 \text{ if } \sum_{i=0}^{3} B_I(ASi) = 0$$

$= 1$ *otherwise*

and

$L_I = 0$ *if* $B_I(ASi) = 0$ *for* $i=0\text{-}3$ *and* $B_I(LSj) = 0$ *for* $j=0\text{-}2$

$= 1$ *otherwise (including* $B_I(LS0) = 255$*)*

At reference point A, the Mux data frame, the interleaved data buffer shall always contain at least the sync byte. The rest of the buffer shall be built in the same manner as the fast buffer, substituting B_I in place of B_F. The length of each mux data frame is K_I bytes, as defined in Figure 65.

The FEC coder shall take in S mux data frames and append R_I FEC redundancy bytes to produce the FEC codeword of length $N_{FEC} = S \times K_I + R_I$ bytes. The FEC output data frames shall contain $N_I = N_{FEC} / S$ bytes, where N_I is an integer. When $S > 1$, then for the S frames in an FEC codeword, the FEC output Data Frame (reference point B) shall partially overlap two mux data frames for all except the last frame, which shall contain the R_I FEC redundancy bytes.

The FEC output data frames are interleaved to a specified interleave depth. The interleaving process delays each byte of a given FEC output data frame a different amount, so that the constellation encoder input data frames will contain bytes from many different FEC data frames. At reference point A in the transmitter, mux data frame 0 of the interleaved data buffer is aligned with the ADSL superframe and mux data frame 0 of the fast data buffer (this is not true at reference point C). At the receiver, the interleaved data buffer will be delayed by (S ' interleave depth ' 250) ms with respect to the fast data buffer, and data frame 0 (containing the CRC bits for the interleaved data buffer) will appear a fixed number of frames after the beginning of the receiver superframe.

3.4.4.1.3 Cyclic redundancy check (CRC)

Two cyclic redundancy checks (CRCs)–one for the fast data buffer and one for the interleaved data buffer–shall be generated for each superframe and transmitted in the first frame of the following superframe. Eight bits per buffer type (fast or interleaved) per superframe allocated to the CRC check bits. These bits are computed from the k message bits using the equation:

$$crc(D) = M(D)\, D^8 \text{ modulo } G(D),$$

where

$$M(D) = m_0 D^{k-1} + m_1 D^{k-2} + \ldots + m_{k-2} D + m_{k-1}$$

is the message polynomial,

$$G(D) = D^8 + D^4 + D^3 + D^2 + 1$$

is the generating polynomial,

$$crc(D) = c_0 D^7 + c_1 D^6 + \ldots + c_6 D + c_7$$

is the check polynomial, and D is the delay operator. That is, CRC is the remainder when $M(D)\, D^8$ is divided by $G(D)$. The CRC check bits are transported in the synchronization bytes (fast and interleaved; 8 bits each) of frame 0 for each data buffer.

The bits (i.e. message polynomials) covered by the CRC include:

Fast data buffer:

- frame 0: ASx bytes (X = 0, 1, 2, 3), LSx bytes (X = 0, 1, 2), followed by any AEX and LEX bytes.
- all other frames: fast byte, followed by ASx bytes (X = 0, 1, 2, 3), LSx bytes (X = 0, 1, 2), and any AEX and LEX bytes.

Interleaved data buffer:

- frame 0: ASx bytes (X = 0, 1, 2, 3), LSx bytes (X = 0, 1, 2), followed by any AEX and LEX bytes.
- all other frames: sync byte, followed by ASx bytes (X = 0, 1, 2, 3), LSx bytes (X = 0, 1, 2), and any AEX and LEX bytes.

Each byte shall be clocked into the CRC least significant bit first.

The number of bits over which the CRC is computed varies with the allocation of bytes to the fast and interleaved data buffers (the numbers of bytes in *ASx* and *LSx* vary according to the $[B_F, B_I]$ pairs; *AEX* is present in a given buffer only if at least one *ASx* is allocated to that buffer; *LEX* is present in a given buffer only if at least one ASx or one *LSx* is allocated to that buffer).

Because of the flexibility in assignment of bearer channels to the fast and interleaved data buffers, CRC field lengths over an ADSL superframe will vary from approximately 67 bytes to approximately 14,875 bytes.

3.4.4.2 Synchronization

If the bit timing base of the input user data streams is not synchronous with the ADSL modem timing base, the input data streams shall be synchronized to the ADSL timing base using the synchronization control mechanism (consisting of synchronization control byte and the AEX and LEX bytes). Forward-error-correction coding shall always be applied to the synchronization control byte(s).

If the bit timing base of the input user data streams is synchronous with the ADSL modem timing base, then the synchronization control mechanism is not needed, and the synchronization control byte shall always indicate "no synchronization action" (see Table 10 and Table 11).

3.4.4.2.1. Synchronization for the fast data buffer.

Synchronization control for the fast data buffer may occur in frames 2 through 33, and 36 through 67 of an ADSL superframe, where the fast byte may be used as the synchronization control byte. No synchronization action shall be taken for those frames for which the fast byte is used for CRC, fixed indicator bits, or EOC.

The format of the fast byte when used as synchronization control for the fast data buffer shall be as given in Table 10.

Table 10- Fast byte format for synchronization

Bits	Designation	Codes
sc7, sc6	ASx bearer channel designator	"00_2" : AS0 "01_2" : AS1 "10_2" : AS2 "11_2" : AS3
sc5, sc4	Synchronization control for the designated ASx bearer channel (see note 3)	"00_2" : no synchronization action "01_2" : add AEX byte to designated ASx bearer channel "11_2" : add AEX and LEX bytes to ASx bearer channel "10_2" : delete last byte from designated ASx bearer channel
sc3, sc2	LSx bearer channel designator (see note 3)	"00_2" : LS0 "01_2" : LS1 "10_2" : LS2 "11_2" : no synchronization action
sc1	Synchronization control for the designated LSx bearer channel	"1_2" : add LEX byte to designated LSx bearer channel "0_2" : delete last byte from designated LSx bearer channel
sc0	Synchronization/EOC designator	"0_2" : perform synchronization control as indicated in sc7-sc1 "1_2" : this byte is part of an EOC frame

ADSL deployments may need to inter-work with DS1 (1.544 Mbit/s) or DS1C (3.152 Mbit/s) rates. The synchronization control option that allows adding up to two bytes to an ASx bearer channel provides sufficient overhead capacity to transport combinations of DS1 or DS1C channels transparently (without interpreting or stripping and regenerating the framing embedded within the DS1 or DS1C). The synchronization control algorithm shall, however, guarantee that the fast byte in some minimum number of frames is available to carry EOC frames, so that a minimum EOC rate (4 kbit/s) may be maintained.

When the data rate of the C channel is 16 kbit/s, the LS0 bearer channel is transported in the LEX byte, using the "add LEX byte to designated LSx channel", with LS0 as the designated channel, every other frame on average.

If the bit timing base of the input bearer channels (ASx, LSx) is synchronous with the ADSL modem timing base, then ADSL systems need not perform synchronization control, (by adding or deleting AEX or LEX bytes to/from the designated ASx and LSx channels). In this case, the synchronization control byte shall indicate "no synchronization action" (i.e., sc7-0 coded "$XX0011X0_2$", with X discretionary).

3.4.4.2.2. Synchronization for the interleaved data buffer

Synchronization control for the interleaved data buffer can occur in frames 1 through 67 of an ADSL superframe, where the sync byte may be used as the synchronization control byte. No synchronization action shall be taken during frame 0, where the sync byte is used for CRC during frames when the LEX byte carries the AOC.

The format of the sync byte when used as synchronization control for the interleaved data buffer shall be as given in Table 11. In the case where no signals are allocated to the interleaved data buffer, the sync byte shall carry the AOC data directly, as shown in Figure 63.

Table 11 - Sync byte format for synchronization

Bits	Designation	Codes
sc7, sc6	ASx bearer channel designator	"00_2" : AS0 "01_2" : AS1 "10_2" : AS2 "11_2" : AS3
sc5, sc4	Synchronization control for the designated ASx bearer channel (see note 3)	"00_2" : no synchronization action "01_2" : add AEX byte to designated ASx bearer channel "11_2" : add AEX and LEX bytes to ASx bearer channel "10_2" : delete last byte from designated ASx bearer channel
sc3, sc2	LSx bearer channel designator	"00_2" : LS0 "01_2" : LS1 "10_2" : LS2 "11_2" : no synchronization action
sc1	Synchronization control for the designated LSx bearer channel	"1_2" : add LEX byte to designated LSx bearer channel "0_2" : delete last byte from designated LSx bearer channel
sc0	Synchronization/AOC designator	"0_2" : perform synchronization control as indicated in sc7-sc1 "1_2" : LEX byte carries ADSL overhead control channel data; synchronization control is allowed for "add AEX" or "delete" as indicated in sc7-sc1

ADSL deployments may need to inter-work with DS1 (1.544 Mbit/s) or DS1C (3.152 Mbit/s) rates. The synchronization control option that allows adding up to two bytes to an ASx bearer channel provides sufficient overhead capacity to transport combinations of DS1 or DS1C channels transparently (without interpreting or stripping and regenerating the framing embedded within the DS1 or DS1C).

When the data rate of the C channel is 16 kbit/s, the LS0 bearer channel is transported in the LEX byte, using the "add LEX byte to designated LSx channel", with LS0 as the designated channel, every other frame on average.

If the bit timing base of the input bearer channels (ASx, LSx) is synchronous with the ADSL modem timing base then ADSL systems need not perform synchronization control by adding or deleting AEX or LEX bytes to/from the designated ASx and LSx channels. In this case, the synchronization control byte shall indicate "no synchronization action". In this case, when framing mode 1 is used, the sc7-0 shall always be coded "$XX0011XX_2$", with X discretionary. When sc0 is set to 1, the LEX byte shall carry AOC. When sc0 is set to 0, the LEX byte shall be coded 00_{16}. The sc0 may be set to 0 only in between transmissions of 5 concatenated and identical AOC messages.

3.4.4.3 Reduced overhead framing

The format described for full overhead framing includes overhead to allow for the synchronization of the seven ASx and LSx bearer channels. When the synchronization function is not required, the ADSL equipment may operate in a reduced overhead mode. This mode retains all the full overhead mode functions except synchronization control.

3.4.4.3.1 Reduced overhead framing with separate fast and sync bytes

The AEX and LEX bytes shall be eliminated from the ADSL frame format, and both the fast and sync bytes shall carry overhead information. The fast byte carries the fast buffer CRC, indicator bits, and EOC messages, and the sync byte carries the interleaved buffer CRC and AOC message. The assignment of overhead functions to fast and sync bytes when using the full overhead framing and when using the reduced overhead framing with separate fast and sync bytes shall be as shown in Table 12.

In the reduced overhead framing with separate fast and sync bytes, the structure of the fast data buffer shall be as shown in Figure 64 with A_F and L_F set to 0. The structure of the interleaved data buffer shall be as shown in Figure 65 with A_I and L_I set to 0.

Table 12 - Overhead functions for framing modes

	Full Overhead Mode		Reduced Overhead Mode	
Frame Number	Fast Sync	Interleave Sync	Fast Sync	Interleave Sync
0	fast CRC	Interleaved CRC	fast CRC	interleaved CRC
1	IB0-7	sync or AOC	IB0-7	AOC
34	IB8-15	sync or AOC	IB8-15	AOC
35	IB16-23	sync or AOC	IB16-23	AOC
all other frames	sync or EOC	sync or AOC	sync or EOC (see NOTE)	AOC

NOTE - In the reduced overhead mode only "no synchronization action" code shall be used.

3.4.4.3.2 Reduced overhead framing with merged fast and sync bytes

In the single latency mode, data is assigned to only one data buffer (fast or interleaved). If data is assigned to only the fast buffer, then only the fast byte shall be used to carry overhead information. If data is assigned only to the interleaved buffer, then only the sync byte shall be used to carry overhead information. Reduced overhead framing with merged fast and sync bytes shall not be used when operating in dual latency mode.

For ADSL systems transporting data using a single data buffer (fast or interleaved), the CRC, indicator, EOC and AOC function shall be carried in a single overhead byte assigned to separate data frames within the superframe structure. The CRC remains in frame 0 and the indicator bits in frames 1, 34, and 35. The AOC and EOC bytes are assigned to alternate pairs of frames. For ADSL equipment operating in single latency mode using the reduced overhead framing with merged fast and sync bytes, the assignment of overhead functions shall be as shown in Table 13.

In the single latency mode using the reduced overhead framing with merged fast and sync bytes, only one data buffer shall be used. If the fast data buffer is used, the structure of the fast data buffer shall be as shown in Figure 64 (with A_F and L_F set to 0) and the interleaved data buffer shall be empty (no sync byte and $K_I = 0$). If the interleaved data buffer is used, the structure of the interleaved data buffer shall be as shown in Figure 65 (with A_I and L_I set to 0) and the fast data buffer shall be empty (no fast byte and $K_F = 0$).

Table 13- Overhead functions for reduced overhead mode with merged fast and sync bytes

Frame Number	(Fast Buffer Only) Fast Byte format	(Interleaved Buffer Only) Sync Byte format
0	Fast CRC	Interleaved CRC
1	IB0-7	IB0-7
34	IB8-15	IB8-15
35	IB16-23	IB16-23
4n+2, 4n+3 with n = 0...16, n ≠8	EOC or sync (see NOTE)	EOC or sync (see NOTE)
4n, 4n+1 with n = 0...16, n ≠0	AOC	AOC
NOTE - In the reduced overhead mode only the "no synchronization action" code shall be used.		

3.4.5 Scramblers

The binary data streams output (LSB of each byte first) from the fast and interleaved data buffers shall be scrambled separately using the following algorithm for both:

$$d'_n = d_n \oplus d'_{n-18} \oplus d'_{n-23}$$

where d_n is the n-th output from the fast or interleaved buffer (i.e., input to the scrambler), and d_n' is the n-th output from the corresponding scrambler. This is illustrated in Figure 66.

These scramblers are applied to the serial data streams without reference to any framing or symbol synchronization. Descrambling in receivers can likewise be performed independent of symbol synchronization.

3.4.6 Forward error correction

The ATU-C shall support downstream transmission with at least any combination of the FEC coding capabilities shown in Table 14.

Table 14 - Minimum FEC coding capabilities for ATU-C

Parameter	Fast buffer	Interleaved buffer
Parity bytes per R-S codeword	R_F = 0,2,4,6,8,10,12,14,16 (see NOTE 2)	R_I = 0,2,4,6,8,10,12,14,16 (see NOTE 1 and NOTE 2)
DMT symbols per R-S codeword	S = 1	S = 1, 2, 4, 8, 16
Interleave depth	not applicable	D = 1, 2, 4, 8, 16, 32, 64
NOTE 1 – R_F can be > 0 only if K_F > 0, and R_I can be > 0 only if K_I > 0 NOTE 2 – R_I shall be an integer multiple of S.		

The ATU-C shall also support upstream transmission with at least any combination of the FEC coding capabilities shown in Table 23.

3.4.6.1 Reed-Solomon coding

R (i.e., R_F or R_I) redundant check bytes $c_0, c_1, \ldots, c_{R-2}, c_{R-1}$ shall be appended to K (i.e., K_f or $S \times K_i$) message bytes $m_0, m_1, \ldots, m_{K-2}, m_{K-1}$ to form a Reed-Solomon code word of size $N = K + R$ bytes. The check bytes are computed from the message byte using the equation:

$$C(D) = M(D)\, D^R \text{ modulo } G(D)$$

where

$$M(D) = m_0 D^{K-1} + m_1 D^{K-2} + \ldots + m_{K-2} D + m_{K-1}$$

is the message polynomial,

$$C(D) = c_0 D^{R-1} + c_1 D^{R-2} + \ldots + c_{R-2} D + c_{R-1}$$

is the check polynomial, and

$$G(D) = P\,(D + a^i)$$

is the generator polynomial of the Reed-Solomon code, where the index of the product runs from $i = 0$ to R-1. That is, $C(D)$ is the remainder obtained from dividing $M(D)\, D^R$ by $G(D)$. The arithmetic is performed in the Galois Field GF(256), where a is a primitive element that satisfies the primitive binary polynomial $x^8 + x^4 + x^3 + x^2 + 1$. A data byte $(d_7, d_6, \ldots, d_1, d_0)$ is identified with the Galois Field element $d_7 a^7 + d_6 a^6 \ldots + d_1 a + d_0$. The number of check bytes is R, and the codeword size N vary.

3.4.6.2 Reed-Solomon Forward Error Correction Superframe Synchronization

When entering the SHOWTIME state after completion of Initialization and Fast Retrain, the ATU shall align the first byte of the first Reed Solomon code-word with the first data byte of DF 0.

3.4.6.3 Interleaving

The Reed-Solomon codewords in the interleaved buffer shall be convolutionally interleaved. The interleaving depth varies, but shall always be a power of 2. Convolutional interleaving is defined by the rule:" Each of the N bytes $B_0, B_1, \ldots, B_{N-1}$ in a Reed-Solomon codeword is delayed by an amount that varies linearly with the byte index. More precisely, byte B_i (with index i) is delayed by (D-1) x i bytes, where D is the interleave depth".

An example for $N = 5$, $D = 2$ is shown in Table 15, where B^j_i denotes the i-th byte of the j-th codeword.

Table 15 - Convolutional interleaving example for $N = 5$, $D = 2$

Inter-leaver input	B^j_0	B^j_1	B^j_2	B^j_3	B^j_4	B^{j+1}_0	B^{j+1}_1	B^{j+1}_2	B^{j+1}_3	B^{j+1}_4
Inter-leaver output	B^j_0	B^{j-1}_3	B^j_1	B^{j-1}_4	B^j_2	B^{j+1}_0	B^j_3	B^{j+1}_1	B^j_4	B^{j+1}_2

With the above-defined rule, and the chosen interleaving depths (powers of 2), the output bytes from the interleaver always occupy distinct time slots when N is odd. When N is even, a dummy byte shall be added at the beginning of the code-word at the input to the interleaver. The resultant odd-length code-word is then convolutionally interleaved, and the dummy byte shall then removed from the output of the interleaver.

3.4.6.4 Support of higher downstream bit rates with S=1/2

With a rate of 4000 data frames per second and a maximum of 255 bytes (maximum R-S code-word size) per data frame, the ADSL downstream line rate is limited to approximately 8 Mbit/s per latency path. The line rate limit can be increased to about 16 Mbit/s for the interleaved path by mapping two RS code-words into one FEC data frame (i.e., by using S=1/2 in the interleaved path). S=1/2 shall be used in the downstream direction only over bearer channel AS0.

When the K_I data bytes per interleaved mux data frame cannot be packed into one RS code-word, i.e., K_I is such that $K_I + R > 255$, the K_I data bytes shall be split into two consecutive RS code-words. When K_I is even, the first and second code-word have the same length $N_i1 = N_i2 = (K_I/2 + R_I)$, otherwise the first code-word is one byte longer than the second, i.e. first code-word has $N_i1 = (K_I + 1)/2 + R_I$ bytes, the second code-word has $N_i2 = (K_I - 1)/2 + R_I$ bytes. For the FEC output data frame, $N_I = N_i1 + N_i2$, with $N_I < 511$ bytes.

The convolutional interleaver requires all code-words to have the same odd length. To achieve the odd code-word length, insertion of a dummy (not transmitted) byte may be required. For S=1/2, the dummy byte addition to the first and/or second code-word at the input of the interleaver shall be as in Table 16.

Table 16 - Dummy byte insertion at interleaver input for $S = 1/2$

N_id1	N_id2	Dummy Byte Insertion Action
odd	odd	no action
even	even	Add one dummy byte at beginning of both code-words
odd	even	Add one dummy byte at the beginning of the second code-word
Even	odd	Add one dummy byte at the beginning of the first code-word and two dummy bytes at the beginning of the second code-word (the de-interleaver shall insert one dummy byte into the de-interleaver matrix on the first byte and the (D + 1)th byte of the corresponding code-word to make the addressing work properly.)

3.4.7 Tone ordering

A DMT time-domain signal has a high peak-to-average ratio (PAR) (its amplitude distribution is almost Gaussian), and large values may be clipped by the digital-to-analog converter. The error signal caused by clipping can be considered as an additive negative impulse for the time sample that was clipped. The clipping error power is almost equally distributed across all tones in the symbol in which clipping occurs. Clipping is therefore most likely to cause errors on those tones that, in anticipation of a higher received SNR, have been assigned the largest number of bits (and therefore have the densest constellations). These occasional errors can be reliably corrected by the FEC coding if the tones with the largest number of bits have been assigned to the interleave buffer.

The numbers of bits and the relative gains to be used for every tone shall be calculated in the ATU-R receiver, and sent back to the ATU-C according to a defined protocol. The pairs of numbers are stored, in ascending order of frequency (or tone number i), in a bit and gain table.

The "tone-ordered" encoding shall first assign the $8*N_F$ bits from the fast data buffer to the tones with the smallest number of bits assigned to them, and then the $8*N_I$ bits from the interleave data buffer to the remaining tones.

All tones shall be encoded with the number of bits assigned to them; one tone may therefore have a mixture of bits from the fast and interleaved buffers.

The ordered bit table b'_i shall be based on the original bit table b_i as follows:

For k = 0 to 15

{

From the bit table, find the set of all i with the number of bits per tone $b_i = k$

Assign b_i to the ordered bit allocation table in ascending order of i

}

A complementary de-ordering procedure should be performed in the ATU-R receiver. It is not necessary, however, to send the results of the ordering process to the receiver because the bit table was originally generated in the ATU-R, and therefore that table has all the information necessary to perform the de-ordering.

Figure 67 and Figure 68 show an example of tone ordering and bit extraction (without and with trellis coding respectively) for a 6-tone DMT case, with N_F=1 and N_I=1 for simplicity.

3.4.8 Constellation encoder (Trellis code version)

Block processing of Wei's 16-state 4-dimensional trellis code is optional in the ITU-T recommendation G.992.1 to improve system performance. An algorithmic constellation encoder shall be used to construct constellations with a maximum number of bits equal to $N_{downmax}$, where $8 \leq N_{downmax} \leq 15$).

3.4.8.1 Bit extraction

Data bytes from the data frame buffer shall be extracted according to a re-ordered bit allocation table b'_i, least significant bit first. Because of the 4-dimensional nature of the code, the extraction is based on pairs of consecutive b'_i, rather than on individual ones, as in the non-trellis-coded case. Furthermore, due to the constellation expansion associated with coding, the bit allocation table, b'_i, specifies the number of coded bits per tone, which can be any integer from 2 to 15. Given a pair (x,y) of consecutive b'_i, $x+y-1$ bits (reflecting a constellation expansion of 1 bit per 4 dimensions, or one half bit per tone) are extracted from the data frame buffer. These $z = x+y-1$ bits (t_z , t_{z-1} , ... , t_1) are used to form the binary word u as shown in Table 17. The tone ordering procedure ensures $x \leq y$. Single-bit constellations are not allowed because they can be replaced by 2-bit constellations with the same average energy. Refer to 0 for the reason behind the special form of the word u for the case $x = 0$, $y > 1$.

Table 17 - Forming the binary word u

Condition	Binary word / comment
$x > 1$, $y > 1$	$u = (t_z, t_{z-1}, ..., t_1)$
$x = 1$, $y \geq 1$	Condition not allowed
$x = 0$, $y > 1$	$u = (t_z, t_{z-1}$, ..., t_2, 0, t_1, 0)
$x = 0$, $y = 0$	Bit extraction not necessary, no message bits being sent
x = 0, y = 1	Condition not allowed
NOTE – t_1 is the first bit extracted from the data frame buffer	

The last two 4-dimensional symbols in the DMT symbol shall be chosen to force the convolutional encoder state to the zero state. For each of these symbols, the 2 LSBs of u are pre-determined, and only $(x+y-3)$ bits shall be extracted from the data frame buffer and shall be allocated to $t_3, t_4, ... t_z$.

3.4.8.2 Bit conversion

The binary word $u=(u_{z'}, u_{z'-1}, ..., u_1)$ determines two binary words $v=(v_{z'-y}, ..., v_0)$ and $w=(w_{y-1}, ..., w_0)$, which are used to look up two constellation points in the encoder constellation table. For the usual case of $x > 1$ and $y > 1$, $z' = z = x+y-1$, and v and w contain x and y bits respectively. For the special case of $x = 0$ and $y > 1$, $z' = z+2 = y+1$, $v = (v_1, v_0) = 0$ and $w=(w_{y-1}, ..., w_0)$. The bits (u_3, u_2, u_1) determine (v_1, v_0) and (w_1, w_0) according to Figure 69.

The convolutional encoder shown in Figure 69 is a systematic encoder (i.e. u_1 and u_2 are passed through unchanged) as shown in Figure 70. The convolutional encoder state (S_3, S_2, S_1, S_0) are used to label the states of the trellis shown in

Figure 72. At the beginning of a DMT symbol period the convolutional encoder state is initialized to *(0, 0, 0, 0)*. The remaining bits of v and w are obtained from the less significant and more significant parts of $(u_{z'}, u_{z'-1}, \ldots, u_4)$, respectively. When $x > 1$ and $y > 1$, $v = (u_{z'-y+2}, u_{z'-y+1}, \ldots, u_4, v_1, v_0)$ and $w = (u_{z'}, u_{z'-1}, \ldots, u_{z'-y+3}, w_1, w_0)$. When $x = 0$, the bit extraction and conversion algorithms have been judiciously designed so that $v_1 = v_0 = 0$. The binary word v is input first to the constellation encoder, and then the binary word w.

In order to force the final state to the zero state (0,0,0,0), the 2 LSBs u_1 and u_2 of the final two 4-dimensional symbols in the DMT symbol are constrained to $u_1 = S_1 \oplus S_3$, and $u_2 = S_2$.

3.4.8.3 Coset partition and trellis diagram

In a trellis code modulation system, the expanded constellation is labeled and partitioned into subsets ("cosets") using a technique called mapping by set-partitioning. The four-dimensional cosets in Wei's code can each be written as the union of two Cartesian products of two 2-dimensional cosets. For example, $C_4^0 = (C_2^0 \cup C_2^1) \times (C_2^2 \cup C_2^3)$. The four constituent 2-dimensional cosets, denoted by C_2^0, C_2^1, C_2^2, C_2^3, are shown in Figure 71.

The encoding algorithm ensures that the 2 least significant bits of a constellation point comprise the index i of the 2-dimensional coset C_2^i in which the constellation point lies. The bits *(v_1, v_0)* and *(w_1, w_0)* are in fact the binary representations of this index.

The three bits *(u_2,u_1,u_0)* are used to select one of the 8 possible four-dimensional cosets. The 8 cosets are labeled C_4^i where i is the integer with binary representation *(u_2,u_1,u_0)*. The additional bit u_3 (see Figure 69) determines which one of the two Cartesian products of 2-dimensional cosets in the 4-dimensional coset is chosen. The relationship is shown in Table 18. The bits (v_1,v_0) and (w_1,w_0) are computed from (u_3,u_2,u_1,u_0) using the linear equations given in Figure 69.

Table 18 - Relation between 4-dimensional and 2-dimensional cosets

4-D Coset	$u_3\ u_2\ u_1\ u_0$	$v_1\ v_0$	$w_1\ w_0$	2-D Cosets
C_4^0	0 0 0 0 1 0 0 0	0 0 1 1	0 0 1 1	$C_2^0 \times C_2^0$ $C_2^3 \times C_2^3$
C_4^4	0 1 0 0 1 1 0 0	0 0 1 1	1 1 0 0	$C_2^0 \times C_2^3$ $C_2^3 \times C_2^0$
C_4^2	0 0 1 0 1 0 1 0	1 0 0 1	1 0 0 1	$C_2^2 \times C_2^2$ $C_2^1 \times C_2^1$
C_4^6	0 1 1 0 1 1 1 0	1 0 0 1	0 1 1 0	$C_2^2 \times C_2^1$ $C_2^1 \times C_2^2$
C_4^1	0 0 0 1 1 0 0 1	0 0 1 1	1 0 0 1	$C_2^0 \times C_2^2$ $C_2^3 \times C_2^1$
C_4^5	0 1 0 1 1 1 0 1	0 0 1 1	0 1 1 0	$C_2^0 \times C_2^1$ $C_2^3 \times C_2^2$
C_4^3	0 0 1 1 1 0 1 1	1 0 0 1	0 0 1 1	$C_2^2 \times C_2^0$ $C_2^1 \times C_2^3$
C_4^7	0 1 1 1 1 1 1 1	1 0 0 1	1 1 0 0	$C_2^2 \times C_2^3$ $C_2^1 \times C_2^0$

Figure 72 shows the trellis diagram based on the finite state machine in Figure 70, and the one-to-one correspondence between (u_2, u_1, u_0) and the 4-dimensional cosets. In the figures, $S=(S_3,S_2,S_1,S_0)$ represents the current state, while $T = (T_3, T_2, T_1, T_0)$ represents the next state in the finite state machine. S is connected to T in the constellation diagram by a branch determined by the values of u_2 and u_1. The branch is labeled with the 4-dimensional coset specified by the values of u_2, u_1 (and $u_0 = S_0$, see Figure 71). To make the constellation diagram more readable, the indices of the 4-dimensional coset labels are listed next to the starting and end points of the branches, rather than on the branches themselves. The leftmost label corresponds to the uppermost branch for each state. The constellation diagram is used when decoding the trellis code by the Viterbi algorithm.

3.4.8.4 Constellation encoder

For a given sub-carrier, the encoder shall select an odd-integer point (X, Y) from the square-grid constellation based on the b bits of either $\{v_{b-1},v_{b-2}, \ldots,v_1,v_0\}$ or $\{w_{b-1},w_{b-2}, \ldots,w_1,w_0\}$. For convenience of description, these b bits are identified with an integer label whose binary representation is $(v_{b-1},v_{b-2}, \ldots,v_1,v_0)$, but the same encoding rules apply also to the w vector. For example, for b=2, the four constellation points are labeled 0,1,2,3 corresponding to (v_1,v_0) = (0,0), (0,1), (1,0), (1,1), respectively (v_0 is the first bit extracted from the buffer).

3.4.8.4.1 Even values of b

For even values of b, the integer values X and Y of the constellation point (X,Y) shall be determined from the b bits $\{v_{b-1}, v_{b-2}, \ldots, v_1, v_0\}$ as follows. X and Y are the odd integers with twos-complement binary representations $(v_{b-1}, v_{b-3}, \ldots, v_1, 1)$ and $(v_{b-2}, v_{b-4}, \ldots, v_0, 1)$, respectively. The most significant bits (MSBs), v_{b-1} and v_{b-2}, are the sign bits for X and Y, respectively.

Figure 74 shows example constellations for $b = 2$ and $b = 4$. (The values of X and Y shown represent the output of the constellation encoder. These values require appropriate scaling such that 1) all constellations regardless of size represent the same RMS energy and 2) by the fine gain scaling before modulation by the IDFT)

The 4-bit constellation can be obtained from the 2-bit constellation by replacing each label n by a 2×2 block of labels as shown in Figure 74. The same procedure can be used to construct the larger even-bit constellations recursively.

The constellations obtained for even values of b are square in shape. The least significant bits $\{v_1, v_0\}$ represent the coset labeling of the constituent 2-dimensional cosets used in the 4-dimensional Wei trellis code.

3.4.8.4.2. Odd values of b, b = 3

Figure 75 shows the constellation for the case b = 3 (the values of X and Y shown represent the output of the constellation encoder. These values require appropriate scaling such that 1) all constellations regardless of size represents the same RMS energy and 2) by the fine gain scaling before modulation by the IDFT).

3.4.8.4.3. Odd values of b, b>3

If b is odd and greater than 3, the 2 MSBs of X and the 2 MSBs of Y are determined by the 5 MSBs of the b bits. Let $c = (b+1)/2$, then X and Y have the twos-complement binary representations $(X_c, X_{c-1}, v_{b-4}, v_{b-6}, \ldots, v_3, v_1, 1)$ and $(Y_c, Y_{c-1}, v_{b-5}, v_{b-7}, v_{b-9}, \ldots, v_2, v_0, 1)$, where X_c and Y_c are the sign bits of X and Y respectively. The relationship between X_c, X_{c-1}, Y_c, Y_{c-1} and $v_{b-1}, v_{b-2}, \ldots, v_{b-5}$ is shown in the Table 19.

Table 19. Determining the top 2 bits of *X* and *Y*

$v_{b-1}, v_{b-2}, \cdots, v_{b-5}$	X_c, X_{c-1}	Y_c, Y_{c-1}
00000	00	00
00001	00	00
00010	00	00
00011	00	00
00100	00	11
00101	00	11
00110	00	11
00111	00	11
01000	11	00
01001	11	00
01010	11	00
01011	11	00
01100	11	11
01101	11	11
01110	11	11
01111	11	11
10000	01	00
10001	01	00
10010	10	00
10011	10	00
10100	00	01
10101	00	10
10110	00	01
10111	00	10
11000	11	01
11001	11	10
11010	11	01
11011	11	10
11100	01	11
11101	01	11
11110	10	11
11111	10	11

Figure 76 shows the constellation for the case b = 5 (the X and Y values are on a ±1, ±3, ... grid. The values of X and Y shown represent the output of the constellation encoder. These values require appropriate scaling such that:

1) all constellations regardless of size represents the same RMS energy and
2) by the fine gain scaling before modulation by the IDFT.

The 7-bit constellation shall be obtained from the 5-bit constellation by replacing each label n by the 2x2 block of labels as shown in Figure 74.

Again, the same procedure shall be used to construct the larger odd-bit constellations recursively. Note also that the least significant bits $\{v_1, v_0\}$ represent the coset labeling of the constituent 2-dimensional cosets used in the 4-dimensional Wei trellis code.

3.4.9 Constellation encoder (No Trellis coding)

An algorithmic constellation encoder shall be used to construct constellations with a maximum number of bits equal to $N_{downmax}$, where $8 \leq N_{downmax} \leq 15$. The constellation encoder shall not use trellis coding with this option.

3.4.9.1 Bit extraction

Data bits from the frame data buffer shall be extracted according to a re-ordered bit allocation table b'_i, least significant bit first. The number of bits per tone, b'_i, can take any non-negative integer values not exceeding $N_{downmax}$, and greater than 1. For a given tone b = b'_i bits are extracted from the data frame buffer, and these bits form a binary word $\{v_{b-1}, v_{b-2}, \ldots, v_1, v_0\}$. The first bit extracted shall be v_0, the LSB.

3.4.10 Gain scaling

For the transmission of data symbols gain scaling, g_i, shall be applied as requested by the ATU-R and possibly updated during Showtime via a bit swap procedure. Only values of g_i equal to zero or within a range of approximately 0.19 to 1.33 (i.e., -14.5 dB to +2.5 dB) may be used. For the transmission of synchronization symbols, no gain scaling shall be applied to any sub-carrier.

Each constellation point, (X_i, Y_i), i.e. complex number $X_i + jY_i$, output from the encoder is multiplied by g_i:

$$Z_i = g_i (X_i + jY_i)$$

3.4.11 Modulation

3.4.11.1 Sub-carriers

The frequency spacing, Δf, between sub-carriers is 4.3125 kHz, with a tolerance of +/– 50 ppm.

3.4.11.1.1 Data sub-carriers

The channel analysis signal allows for a maximum of 255 carriers (at frequencies $n\Delta f$, n = 1 to 255) to be used. The lower limit of n depends on both the duplexing and service options selected. For example, for ADSL above POTS service option, if overlapped spectrum is used to separate downstream and upstream signals, then the lower limit on n is determined by the POTS splitting filters; if frequency division multiplexing (FDM) is used, the lower limit is set by the downstream-upstream separation filters.

3.4.11.1.2 Pilot

Carrier #N_{pilot} (f_{pilot} = 4.3125 x N_{pilot} kHz) shall be reserved for a pilot; that is $b(N_{pilot}) = 0$ and $g(N_{pilot}) = 1$.

The data modulated onto the pilot sub-carrier shall be a constant {0,0}. Use of this pilot allows resolution of sample timing in a receiver modulo-8 samples. Therefore a gross timing error that is an integer multiple of 8 samples could still persist after a micro-interruption (e.g., a temporary short- circuit, open circuit or severe line hit); correction of such timing errors is made possible by the use of the synchronization symbol.

3.4.11.1.3 Nyquist frequency

The carrier at the Nyquist frequency (#256) shall not be used for user data and shall be real valued.

3.4.11.1.4 DC

The carrier at DC (#0) shall not be used, and shall contain no energy.

3.4.11.2 Modulation by the inverse discrete Fourier transform (IDFT)

The modulating transform defines the relationship between the 512 real values x_n and the Z_i.

$$x_n = \sum_{i=0}^{511} \exp(\frac{j\pi ni}{256})\ Z_i \quad \text{for n = 0 to 511}$$

The constellation encoder and gain scaling generate only 255 complex values of Z_i. In order to generate real values of x_n the input values (255 complex values plus zero at DC and one real value for Nyquist if used) shall be augmented so that the vector Z has Hermitian symmetry. That is,

$$Z_i = \text{conj}\,(Z'_{512-i}) \quad \text{for } i = 257 \text{ to } 511$$

3.4.11.3 Synchronization symbol

The synchronization symbol permits recovery of the frame boundary after micro-interruptions that might otherwise force retraining. The data symbol rate, f_{symb} = 4 kHz, the carrier separation, Δf= 4.3125 kHz, and the IDFT size, N = 512, are such that a cyclic prefix of 40 samples could be used. That is,

$$(512 + 40) \times 4.0 = 512 \times 4.3125 = 2208$$

The cyclic prefix shall, however, be shortened to 32 samples, and a synchronization symbol (with a nominal length of 544 samples) is inserted after every 68 data symbols. That is,

$$(512 + 32) \times 69 = (512 + 40) \times 68$$

The data pattern used in the synchronization symbol shall be the pseudo-random sequence PRD, (d_n, for n = 1 to 512) defined by

$d_n = 1$ for n = 1 to 9

$d_n = d_{n-4} \oplus d_{n-9}$ for n =10 to 512

The first pair of bits (d_1 and d_2) shall be used for the DC and Nyquist sub-carriers (the power assigned to them is zero, so the bits are effectively ignored). The first and second bits of subsequent pairs are then used to define the X_i and Y_i for i = 1 to 255 as shown in Table 20.

Table 20. Mapping of two data bits into a 4QAM constellation

$d2i+1, d2i+2$	Decimal label	X_i, Y_i
0 0	0	+ +
0 1	1	+ –
1 0	2	– +
1 1	3	– –

The period of the PRD is only 511 bits, so d512 shall be equal to d1. The d1 - d9 shall be re-initialized for each synchronization symbol, so each symbol uses the same data. Bits 129 and 130, which modulate the pilot carrier, shall be overwritten by {0,0}: generating the {+,+} constellation.

The minimum set of sub-carriers to be used is the set used for data transmission (i.e., those for which $b_i > 0$). The data modulated onto each sub-carrier shall be as defined above; it shall not depend on which sub-carriers are used.

3.4.12. Cyclic prefix

The last 32 samples of the output of the IDFT (x_n for n = 480 to 511) shall be prepended to the block of 512 samples and read out to the digital-to-analog converter (DAC) in sequence. That is, the subscripts, n, of the DAC samples in sequence are 480.....511,0.....511.

3.4.13. Transmitter dynamic range

The transmitter includes all analog transmitter functions: the D/A converter, the anti-aliasing filter, the hybrid circuitry, and the high-pass part of the POTS or ISDN splitter.

3.4.13.1. Maximum clipping rate

The maximum output signal of the transmitter shall be such that the signal shall be clipped no more than 0.00001% of the time.

3.4.13.2. Noise/Distortion floor

The signal to noise plus distortion ratio of the transmitted signal in a given sub-carrier is specified as the ratio of the rms value of the tone in that sub-carrier to the rms sum of all the non-tone signals in the 4.3125 kHz frequency band centered on the sub-carrier frequency. This ratio is measured for each sub-carrier used for transmission using a MultiTone Power Ratio (MTPR) test as shown in Figure 77.

Over the transmission frequency band, the MTPR of the transmitter in any sub-carrier shall be no less than ($3N_{\text{downi}}$ + 20)dB, where N_{downi} is defined as the size of the constellation (in bits) to be used on sub-carrier i. The minimum transmitter MTPR shall be at least 38dB (corresponding to an N_{downi} of 6) for any sub-carrier.

Signals transmitted during normal initialization and data transmission cannot be used for this test because the DMT symbols have a cyclic prefix appended, and the PSD of a non-repetitive signal does not have nulls at any sub-carrier frequencies. A gated FFT-based analyzer could be used, but this would measure both the non-linear distortion and the linear distortion introduced by the transmit filter. Therefore this test will require that the transmitter be programmed with special software probably to be used during development only.

3.5 ATU-R Functional Characteristics

An ATU-R may support STM transmission or ATM transmission or both framing modes that shall be supported, depend upon the ATU-R being configured for either STM or ATM transport. If framing mode k is supported, then modes k-1, ..., 0 shall also be supported.

During initialization, the ATU-C and ATU-R shall indicate a framing mode number 0, 1, 2 or 3 which they intend to use. The lowest indicated framing mode shall be used.

An ATU-R may support reconstruction of a Network Timing Reference (NTR) from the downstream indicator bits.

3.5.1 STM Transmission Protocol Specific functionalities

3.5.1.1 ATU-R input and output V interfaces for STM transport

The functional data interfaces at the ATU-R are shown in Figure 78. Output interfaces for the high-speed downstream simplex bearer channels are designated AS0 through AS3; input–output interfaces for the duplex bearer channels are designated LS0 through LS2. There may also be a functional interface to transport operations, administration and maintenance (OAM) indicators from the CI to the ATU-R; this interface may physically be combined with the LS0 interface.

3.5.1.2 Downstream simplex channels – Transceiver bit rates

The simplex channels are transported in the downstream direction only; therefore their data interfaces at the ATU-R operate only as outputs.

3.5.1.3 Duplex channels – Transceiver bit rates

The duplex channels are transported in both directions, so the ATU-R shall provide both input and output data interfaces.

3.5.1.4 Framing Structure for STM transport

An ATU-R configured for STM transport shall support the full overhead framing structure 0. The support of full overhead framing structure 1 and reduced overhead framing structures 2 and 3 is optional.

Preservation of T-R interface byte boundaries (if present) at the U-R interface may be supported for any of the U-R interface framing structures.

An ATU-R configured for STM transport may support reconstruction of a Network Timing Reference (NTR).

3.5.2 ATM Transport Protocol Specific functionalities

3.5.2.1 ATU-R input and output V interfaces for ATM transport

The ATU-R input and output T interfaces are identical to the ATU-C input and output interfaces, as shown in Figure 79.

3.5.2.2 ATM Cell specific functionalities

The ATM cell specific functionalities performed at the ATU-R shall be identical to the ATM cell specific functionalities performed at the ATU-C.

3.5.2.3 Framing Structure for ATM transport

An ATU-R configured for ATM transport shall support the full overhead framing structures 0 and 1.

The ATU-R transmitter shall preserve T-R interface byte boundaries (explicitly present or implied by ATM cell boundaries) at the U-R interface, independent of the U-R interface framing structure.

An ATU-R configured for ATM transport may support reconstruction of a Network Timing Reference (NTR).

To ensure framing structure 0 interoperability between an ATM ATU-R and an ATM cell TC plus an STM ATU-C (i.e., ATM over STM), the following shall apply:

- An STM ATU-C transporting ATM cells and not preserving V-C byte boundaries at the U-C interface shall indicate during initialization that frame structure 0 is the highest frame structure supported;
- An STM ATU-C transporting ATM cells and preserving V-C byte boundaries at the U-C interface shall indicate during initialization that frame structure 0, 1, 2 or 3 is the highest frame structure supported, as applicable to the implementation;
- An ATM ATU-R receiver operating in framing structure 0 can not assume that the ATU-C transmitter will preserve V-C interface byte boundaries at the U-C interface and shall therefore perform the cell delineation bit-by-bit.

3.5.3 Network timing reference

If the ATU-C has indicated that it will use indicator bits 20 to 23 to transmit the change of phase offset, the ATU-R may deliver the 8 kHz signal to the T-R interface

3.5.4 Framing

Framing of the upstream signal (ATU-R transmitter) closely follows the downstream framing (ATU-C transmitter), but with the following exceptions:

- There are no ASx channels and no AEX byte;
- A maximum of three channels exist, so that only three B_F, B_I pairs are specified;
- The minimum RS FEC coding parameters and interleave depth differ (see Table 23);
- Four bits of the fast and sync bytes are unused (corresponding to the bit positions used by the ATU-C transmitter to specify synchronization control for the ASx channels) (see Table 21 and Table 22).
- The four indicator bits for NTR transport are not used in upstream direction.

Two types of framing are defined: full overhead and reduced overhead. Furthermore, two versions of full overhead and two versions of reduced overhead are defined. The four resulting framing structures are defined as for the ATU-C and are referred to as framing structures 0, 1, 2 and 3.

Requirements for framing structures to be supported, depend upon the ATU-R being configured for either STM or ATM transport.

Outside the ASx/LSx serial interfaces data bytes are transmitted MSB first in accordance with ITU-T Recommendations G.703, G.707, I.361, and I.432. All serial processing in the ADSL frame (e.g., CRC, scrambling, etc.) shall, however, be performed LSB first, with the outside world MSB considered by the ADSL as LSB. As a result, the first incoming bit (outside world MSB) will be the first processed bit inside the ADSL (ADSL LSB)

3.5.4.1 Data symbols

The ATU-R transmitter is functionally similar to the ATU-C transmitter, except that up to three duplex data channels are synchronized to the 4 kHz ADSL DMT symbol rate (instead of up to four simplex and three duplex channels as is the case for the ATU-C). The ATU-R transmitter and its associated reference points for data framing are shown in Figure 55 and Figure 56.

3.5.4.1.1 Superframe structure

The superframe structure of the ATU-R transmitter is identical to that of the ATU-C transmitter, shown in Figure 61.

The ATU-R shall support the indicator bits. The indicator bits, ib20-23, shall not transport NTR in the upstream direction and shall be set to 1.

3.5.4.1.2. Frame structure (with full overhead)

Each data frame shall be encoded into a DMT symbol. As specified for the ATU-C shown in Figure 61, each frame is composed of a fast data buffer and an interleaved data buffer, and the frame structure has a different appearance at each of the reference points (A, B, and C). The bytes of the fast data buffer shall be clocked into the constellation encoder first, followed by the bytes of the interleaved data buffer. Bytes are clocked least significant bit first.

The assignment of bearer channels to the fast and interleaved buffers shall be configured during initialization with the exchange of a (B_F,B_I) pair for each data stream, where B_F designates the number of bytes of a given data stream to allocate to the fast buffer, and B_I designates the number of bytes allocated to the interleaved data buffer.

The three possible (B_F,B_I) pairs are B_F(LSx), B_I(LSx) for X = 0, 1 and 2, for the duplex channels; they are specified as for the ATU-C.

3.5.4.1.2.1 Fast data buffer

The frame structure of the fast data buffer is the same as that specified for the ATU-C with the following exceptions:

- ASx bytes do not appear;
- The AEX byte does not appear;

The following shall hold for the parameters shown in Figure 80.

$C_F(LS0)$ = 0 if $B_F(LS0) = 255$ (11111111_2)

= $B_F(LS0)$ otherwise

L_F = 0 if $B_F(LS0) = B_F(LS1) = B_F(LS2) = 0$

= 1 otherwise

K_F = $1 + C_F(LS0) + B_F(LS1) + B_F(LS2) + L_F$

N_F = $K_F + R_F$

where R_F = number of upstream FEC redundancy bytes in fast path.

At reference point A (the mux data frame) in Figure 55 and Figure 56 the fast buffer always contains at least the fast byte. This is followed by B_F(LS0) bytes of channel LS0, then B_F(LS1) bytes of channel LS1, and B_F(LS2) bytes of channel LS2, and if any B_F(LSx) is non-zero, a LEX byte.

When B_F(LS0) = 255 (11111111_2), no separate bytes are included for the LS0 channel. Instead, the 16 kbit/s C channel shall be transported in every other LEX byte on average, using the synchronization byte to denote when to add the LEX byte to the LS0 bearer channel.

R_F FEC redundancy bytes shall be added to the mux data frame (reference point A) to produce the FEC output data frame (reference point B), where R_F is given in the C-RATES1 signal options received from the ATU-C during initialization.

Because the data from the fast data buffer is not interleaved, the constellation encoder input data frame (reference point C) is identical to the FEC output data frame (reference point B).

3.5.4.1.2.2 Interleaved data buffer

The frame structure of the interleaved data buffer is shown in Figure 81 for the three reference points that are defined in Figure 55 and Figure 56. This structure is the same as that specified for the ATU-C, with the following exceptions:

- ASx bytes do not appear;
- the AEX byte does not appear;

The following shall hold for the parameters shown in Figure 81:

$C_I(LS0)$ $= 0$ if $B_I(LS0) = 255$ (11111111_2)

$= B_I(LS0)$ otherwise

L_I $= 0$ if $B_I(LS0) = B_I(LS1) = B_I(LS2) = 0$

$= 1$ otherwise

K_I $= 1 + C_I(LS0) + B_I(LS1) + B_I(LS2) + L_I$

N_I $= (S \cdot K_I + R_I) / S$

where R_I = number of upstream FEC redundancy bytes in interleaved path and S = number of mux data frames per FEC codeword.

3.5.4.1.3 Cyclic redundancy check (CRC)

Two cyclic redundancy checks (CRCs) – one for the fast data buffer and one for the interleaved data buffer – are generated for each superframe and transmitted in the first frame of the following superframe. Eight bits per buffer type (fast or interleaved) per superframe are allocated to the CRC check bits. These bits are computed from the k message bits using the equation:

$$\text{crc}(D) = M(D)\, D^8 \text{ modulo } G(D),$$

where

$$M(D) = m_0 D^{k-1} + m_1 D^{k-2} + ... + m_{k-2} D + m_{k-1}$$

is the message polynomial,

$$G(D) = D^8 + D^4 + D^3 + D^2 + 1$$

is the generating polynomial,

$$\text{crc}(D) = c_0\, D^7 + c_1 D^6 + ... + c_6\, D + c_7$$

The CRC bits are transported in the fast byte (8 bits) of frame 0 in the fast data buffer, and the sync byte (8 bits) of frame 0 in the interleaved data buffer. The bits covered by the CRC include;

- for the fast data buffer:
  - *frame 0:* LSx bytes ($X = 0, 1, 2$), followed by the LEX byte;
  - *all other frames:* fast byte, followed by LSx bytes ($X = 0, 1, 2$), and LEX byte.
- for the interleaved data buffer:
  - *frame 0:* LSx bytes ($X = 0, 1, 2$), followed by the LEX byte;
  - *all other frames:* sync byte, followed by LSx bytes ($X = 0, 1, 2$), and LEX byte.

Each byte shall be clocked into the CRC least significant bit first.

The CRC-generating polynomial, and the method of generating the CRC byte are the same as for the downstream data.

3.5.4.2 Synchronization

If the bit timing base of the input user data streams is not synchronous with the ADSL modem timing base the input data streams shall be synchronized to the ADSL timing base using the synchronization control mechanism (consisting of synchronization control byte and the LEX byte). Forward-error-correction coding shall always be applied to the synchronization control byte(s).

If the bit timing base of the input user data streams is synchronous with the ADSL modem timing base then the synchronization control mechanism is not needed. The synchronization control byte shall always indicate "no synchronization action".

3.5.4.2.1. Synchronization for the fast data buffer

Synchronization control for the fast data buffer can occur in frames 2 through 33 and 36 through 67 of an ADSL superframe, where the fast byte may be used as the synchronization control byte. No synchronization action is to be taken for those frames in which the fast byte is used for CRC, fixed indicator bits, or EOC. The format of the fast byte when used as synchronization control for the fast data buffer shall be as given in Table 21.

In the case where no signals are allocated to the interleaved data buffer, the sync byte carries the AOC data directly as shown in Figure 63.

Table 21 - Fast byte format for synchronization

Bit	Application	Specific usage
sc7-sc4	not used	set to "0_2" until specified otherwise
sc3, sc2	LSx channel designator	"00_2" : channel LS0 "01_2" : channel LS1 "10_2" : channel LS2 "11_2" : no synchronization action
sc1	Synchronization control for the designated LSx channel	"1_2" : add LEX byte to designated LSx channel "0_2" : delete last byte from designated LSx channel
sc0	Synchronization/EOC designator	"0_2" : perform synchronization control as indicated in sc7-sc1 "1_2" : this byte is part of an EOC frame

If the bit timing base of the input bearer channels (LSx) is synchronous with the ADSL modem timing base then ADSL systems need not perform synchronization control by adding or deleting LEX bytes to/from the designated LSx channels. The synchronization control byte shall indicate "no synchronization action" (i.e., sc7-0 coded "$000011X0_2$", with X discretionary).

When the data rate of the C channel is 16 kbit/s, the LS0 bearer channel shall be transported in the LEX byte, using the "add LEX byte to designated LSx channel", with LS0 as the designated channel, every other frame on average.

3.5.4.2.2. Synchronization for the interleaved data buffer

Synchronization control for the interleaved data buffer can occur in frames 1 through 67 of an ADSL superframe, where the sync byte may be used as the synchronization control byte. No synchronization action shall be taken during frame 0, where the sync byte is used for CRC, and frames when the LEX byte carries the AOC.

The format of the sync byte when used as synchronization control for the interleaved data buffer shall be as given in Table 22. In the case where no signals are allocated to the interleaved data buffer, the sync byte shall carry the AOC data directly, as shown in Figure 63.

Table 22 - Sync byte format for synchronization

Bit	Application	Specific usage
sc7-sc4	not used	Set to "0_2" until specified otherwise
sc3, sc2	LSx channel designator	"00_2" : channel LS0 "01_2" : channel LS1 "10_2" : channel LS2 "11_2" : no synchronization action
sc1	Synchronization control for the designated LSx channel	"1_2" : add LEX byte to designated LSx channel "0_2" : delete last byte from designated LSx channel
sc0	Synchronization/AOC designator	"0_2" : perform synchronization control as indicated in sc3-sc1 "1_2" : LEX byte carries ADSL overhead control channel data; a delete synchronization control may be allowed as indicated in sc3-sc1

When the data rate of the C channel is 16 kbit/s, the LS0 bearer channel shall be transported in the LEX byte, using the "add LEX byte to designated LSx channel", with LS0 as the designated channel, every other frame on average.

If the bit timing base of the input bearer channels (LSx) is synchronous with the ADSL modem timing base then ADSL systems need not perform synchronization control by adding or deleting LEX bytes to/from the designated LSx channels, and the synchronization control byte shall indicate "no synchronization action". In this case, and when framing structure 1 is used, the sc7-0 shall always be coded "000011XX$_2$", with X discretionary. When sc0 is set to 1, the LEX byte shall carry AOC. When sc0 is set to 0, the LEX byte shall be coded 00_{16}. The sc0 may be set to 0 only in between transmissions of 5 concatenated and identical AOC messages.

3.5.4.3 Reduced overhead framing

The format described in 2.5.4.1.2 for full overhead framing includes overhead to allow for the synchronization of three LSx bearer channels. When the synchronization function described in 2.5.4.2 is not required, the ADSL equipment may operate in a reduced overhead mode. This mode retains all the full overhead mode functions except synchronization control. When using the reduced overhead framing, the framing structure shall be as defined in 2.4.4.3.1 (when using separate fast and sync bytes) or2.4.4.3.2 (when using merged fast and sync bytes)

3.5.5 Scramblers

The data streams output from the fast and interleaved buffers shall be scrambled separately using the same algorithm as for the downstream signal.

3.5.6 Forward error correction

The upstream data shall be Reed-Solomon coded and interleaved using the same algorithm as for the downstream data.

The ATU-R shall support upstream transmission with at least any combination of the FEC coding capabilities shown in Table 23.

Table 23 - Minimum FEC coding capabilities for ATU-R

Parameter	Fast buffer	Interleaved buffer
Parity bytes per R-S codeword	R_F = 0,2,4,6,8,10,12,14,16 (see NOTE 2)	R_I = 0,2,4,6,8,10,12,14,16 (see NOTE 1 and NOTE 2)
DMT symbols per R-S codeword	S = 1	S = 1, 2, 4, 8, 16
Interleave depth	not applicable	D = 1, 2, 4, 8
NOTE 1 - R_F can be >0 only if $K_F > 0$ and R_I can be > 0 only if $K_I > 0$ NOTE 2 - R_I shall be an integer multiple of S.		

The ATU-R shall also support upstream transmission with at least any combination of the FEC coding capabilities shown in Table 14.

3.5.7 Tone ordering

The tone-ordering algorithm shall be the same as for the downstream data.

3.5.8 Constellation encoder - Trellis version

Block processing of Wei's 16-state 4-dimensional trellis code to improve system performance is optional. An algorithmic constellation encoder shall be used to construct constellations with a maximum number of bits equal to N_{upmax}, where $8 \leq N_{upmax} \leq 15$.

The encoding algorithm shall be the same as that used for downstream data (with the substitution of the constellation limit of N_{upmax} for $N_{downmax}$).

3.5.9 Constellation encoder – Uncoded version

An algorithmic constellation encoder shall be used to construct constellations with a maximum number of bits equal to N_{upmax}, where $8 \leq N_{upmax} \leq 15$. The encoding algorithm is the same as that used for downstream data (with the substitution of the constellation limit of N_{upmax} for $N_{downmax}$). The constellation encoder shall not use trellis coding with this option.

3.5.10 Gain scaling

For the transmission of data symbols gain scaling, g_i, shall be applied as requested by the ATU-C and possibly updated during Showtime via the bit swap procedure. Only values of g_i equal to 0 or within a range of approximately 0.19 to 1.33 (i.e., -14.5 dB to +2.5 dB) may be used. For the transmission of synchronization symbols, no gain scaling shall be applied to any sub-carrier.

Each constellation point, (X_i,Y_i), i.e. complex number, $X_i + jY_i$, output from the encoder is multiplied by g_i:

$$Z_i = g_i (X_i + jY_i)$$

3.5.11. Modulation

Frequency spacing, Δf, between sub-carriers shall be 4.3125 kHz with a tolerance of +/– 50 ppm.

3.5.11.1. Sub-carriers

3.5.11.1.1. Data sub-carriers

The channel analysis signal allows for a maximum of 31 carriers (at frequencies $n\Delta f$) to be used. The range of n depends on the service option selected. For example, for ADSL above POTS the lower limit is set by the POTS/ADSL splitting filters; the upper limit is set by the transmit and receive band-limiting filters, and shall be no greater than 31. The cut-off frequencies of these filters are at the discretion of the manufacturer because the range of usable n is determined during the channel estimation.

3.5.11.1.2 Nyquist frequency

The sub-carrier at the Nyquist frequency shall not be used for user data and shall be real valued.

3.5.11.1.3 DC

The sub-carrier at DC (#0) shall not be used, and shall contain no energy.

3.5.11.2. Synchronization symbol

The synchronization symbol permits recovery of the frame boundary after micro-interruptions that might otherwise force retraining.

The data symbol rate, f_{symb} = 4 kHz, the sub-carrier separation, Δf = 4.3125 kHz, and the IDFT size, N = 64, are such that a cyclic prefix of 5 samples could be used. That is,

$$(64 + 5) * 4.0 = 64 * 4.3125 = 276$$

The cyclic prefix shall, however, be shortened to 4 samples, and a synchronization symbol (with a nominal length of 68 samples) inserted after every 68 data symbols. That is,

$$(64 + 4) * 69 = (64 + 5) * 68$$

The minimum set of sub-carriers to be used is the set used for data transmission (i.e., those for which $b_i > 0$); sub-carriers for which $b_i = 0$ may be used at a reduced PSD. The data modulated onto each sub-carrier shall be as defined above; it shall not depend on which sub-carriers are used.

3.5.12. Transmitter dynamic range

The transmitter includes all analog transmitter functions: the D/A converter, the anti-aliasing filter, the hybrid circuitry, and the POTS splitter.

3.5.12.1. Maximum clipping rate

The maximum output signal of the transmitter shall be such that the signal shall be clipped no more than 0.00001% of the time.

3.5.12.2 Noise/Distortion floor

The signal to noise plus distortion ratio of the transmitted signal in a given sub-carrier ($(S/N+D)_i$) is specified as the ratio of the rms value of the full-amplitude tone in that sub-carrier to the rms sum of all the non-tone signals in the 4.3125 kHz frequency band centered on the sub-carrier frequency. This ratio is measured for each sub-carrier used for transmission using a Multi-Tone Power Ratio (MTPR) test as shown in Figure 77.

Over the transmission frequency band, the MTPR of the transmitter in any sub-carrier shall be no less than $(3N_{upi} + 20)$ dB, where N_{upi} is defined as the size of the constellation (in bits) to be used on sub-carrier i. The transmitter MTPR shall be +38dB (corresponding to an N_{upi} of 6) for any sub-carrier.

Signals transmitted during normal initialization and data transmission cannot be used for this test because the DMT symbols have a cyclic prefix appended, and the PSD of a non-repetitive signal does not have nulls at any sub-carrier frequencies. A gated FFT-based analyzer could be used, but this would measure both the non-linear distortion and the linear distortion introduced by the transmit filter. Therefore this test will require that the transmitter be programmed with special software, probably to be used during development only.

4.- Use of SMCCC and PMCCC for wired communications as xDSL.

We now describe the use of SMCCC and PMCCC for xDSL systems and, in particular, apply it to the case of ADSL transceivers. The use for other xDSL systems and other data communications systems is straightforward. It should be noted that the appended claims are not intended to be limited to the particular application of ADSL.

An SMCCC is formed by two (or more) constituent systematic encoders joined through an interleaver. The input information bits feed the first encoder and, after having been scrambled by the interleaver, enter the second encoder. A code word of a serial concatenated code comprises of the input bits to the first encoder followed by the parity check bits of both

encoders. SMCCC achieves near-Shannon-limit error correction performance. Here we describe the proposed encoder, the decoder and some simulation results.

4.1- Parallel Multiple Convolutional Concatenated Codes.

A PMCCC encoder is formed by two (or more) constituent systematic encoders joined through one or more interleavers. The input information bits feed the first encoder and, after having been scrambled by the interleaver, enter the second encoder. A code word of a parallel concatenated code comprises of the input bits to the first encoder followed by the parity check bits of both encoders. Here we present the proposed encoder, the decoder and some simulation results. The disadvantage of the PMCCC is that it has a floor-error around 10^{-6}. This could be improved with a good interleaver design, but using a large number of iterations.

4.2.1- Parallel Multiple Convolutional Concatenated Codes Encoder.

A PMCCC encoder comprises of two parallel concatenated recursive systematic convolutional encoders separated by an interleaver. The encoders are arranged in a "parallel concatenation". In a preferred embodiment, the concatenated recursive systematic convolutional encoders may be identical.

Figure 82 represents the proposed encoder. The input is a block of information bits. The two encoders generate parity symbols (u_0 and u'_0) from two simple recursive convolutional codes. The key innovation of this technique is an interleaver "t", which permutes the original information bits before input to the second encoder. The permutation performed by the interleaver allows those input sequences for which one encoder produces low-weight codewords to usually cause the other encoder to produce high-weight codewords. Thus, even though the constituent codes are individually weak, the combination is surprisingly powerful. The resulting code has features similar to a "random" block code.

In this way, we have the information symbols (u_1 and u_2) and two redundant symbols (u_0 and u'_0). With this redundancy it is possible to reach longer loops and to reduce the PAR, at the cost of a slight increase of the constellation encoder.

In the Figure 83 we have presented the conversion that we propose, taking into account the new parity bit.

4.2.2- Parallel Multiple convolutional Concatenated Codes Decoder.

In Figure 84 we present the decoder, that uses an iterative technique, with two soft-decision input/output trellis decoder in each decoding state. The Maximum-a-Posteriori (MAP) Trellis decoder provides the soft output result suitable for PMCCC decoding.

The first decoder should deliver a soft output to the second decoder. The logarithm of the Likelihood Ratio (LLR) of a bit decision is the soft decision information output by the MAP decoder.

Let u_k be the binary random variable taking values in $\{0,1\}$, representing the sequence of information bits $u=(u_1,\ldots u_n)$. The optimum decision algorithm on the kth bit u_k is based on the conditional log-likelihood ratio L_k

$$L_k = \log \frac{P(u_k = 1|y)}{P(u_k = 0|y)} = \log \frac{\sum_{u:u_k=1} \prod_{i=0}^{2} P(y_i|u)}{\sum_{u:u_k=0} \prod_{i=0}^{2} P(y_i|u)} \quad (107)$$

where $P(u_i)$ are the a priori probabilities.

Using Bayes' rule and the following approximation:

$$P(u|y_i) \approx \prod_{k=1}^{n} \overline{P}_i(u_k) \quad (108)$$

The MAP algorithm approximates a nonseparable distribution with a separable one. It is possible to separate $P(u|y_i)$

$$\overline{P}_i(u_k) = \frac{e^{u_k \bar{L}_{ik}}}{1 + e^{\bar{L}_{ik}}} \quad (109)$$

$$L_k = f(y_1, \overline{L}_0, \overline{L}_2, k) + \overline{L}_{0k} + \overline{L}_{2k} \quad (110)$$

For binary modulation:

$$\overline{L}_{0k} = \frac{2Ay_{0k}}{\sigma^2} \quad (111)$$

$$f(y_1, \overline{L}_0, \overline{L}_2, k) = \log \frac{\sum_{u:u_k=1} P(y_1|u) \prod_{j \neq k} e^{u_j(\overline{L}_{0j} + \overline{L}_{2j})}}{\sum_{u:u_k=0} P(y_1|u) \prod_{j \neq k} e^{u_j(\overline{L}_{0j} + \overline{L}_{2j})}} \quad (112)$$

and similarly:

$$L_k = f(y_2, \overline{L}_0, \overline{L}_1, k) + \overline{L}_{0k} + \overline{L}_{1k} \quad (113)$$

$$f(y_2, \overline{L}_0, \overline{L}_1, k) = \log \frac{\sum_{u:u_k=1} P(y_2|u) \prod_{j \neq k} e^{u_j(\overline{L}_{0j} + \overline{L}_{1j})}}{\sum_{u:u_k=0} P(y_2|u) \prod_{j \neq k} e^{u_j(\overline{L}_{0j} + \overline{L}_{1j})}} \quad (114)$$

A solution to this equation is:

$$\overline{L}_{1k} = f(y_1, \overline{L}_0, \overline{L}_2, k) \quad (115)$$

$$\overline{L}_{2k} = f(y_2, \overline{L}_0, \overline{L}_1, k) \quad (116)$$

for $k=1,2,...,n$. The final decision is based on:

$$L_k = \overline{L}_{0k} + \overline{L}_{2k} \quad (117)$$

which is passed through a hard limiter with zero threshold.

The nonlinear equations can be solve using the iterative procedure:

$$\overline{L}_{1k}^{(m+1)} = \alpha_1^{(m)} f(y_1, \overline{L}_0, \overline{L}_2^{(m)}, k) \quad (118)$$

$$\overline{L}_{2k}^{(m+1)} = \alpha_2^{(m)} f(y_2, \overline{L}_0, \overline{L}_1^{(m)}, k) \quad (119)$$

The recursion can be started with the initial condition:

$$\overline{L}_1^{(0)} = \overline{L}_2^{(0)} = \overline{L}_0 \quad (120)$$

For each iteration $\alpha_1^{(m)}$ and $\alpha_2^{(m)}$ can be optimized or set to 1 for simplicity.

4.2.3. Design of the interleaver for PMCCC.

In a PMCCC the interleaver establishes a relationship between portions of a codeword. It is generally assumed that when a PMCCC decoder is operating at low bit error rates, error sequences have small Hamming weights. From this, and properties of PMCCC, a mathematical structure is possible to developed for interleaver design, permitting the identification of quantitatively optimal interleaver. Simulations show the math captures some but not all the essential characteristics of a successful interleaver. Modifying a random interleaver according to some mathematical ideas gives excellent simulation results.

The function of the interleaver in the PMCCC is to assure that at least one of the codeword components has high Hamming weight. For a better PMCCC, we can design an interleaver of permutation length p that maximizes the minimum Hamming weight generated by weight two inputs. This requires maximizing:

$$\sigma_\pi \equiv \min_{i,j} |j - i| + |\pi(j) - \pi(i)| \quad 1 \leq i, j \leq p \quad (121)$$

where π is the interleaver function. It is also possible to replace the sum with the maximum of:

$$s_\pi \equiv \min_{i,j} |j - i| \vee |\pi(j) - \pi(i)| \quad 1 \le i,j \le p \quad (122)$$

An alternate method for interleaver design is to disperse symbols as widely as possible in a "constellation way". One effective method is to choose s_1 and s_2 and generate π one point at a time. For each $i \in [1,p]$, taken sequentially, random values are considered for $\pi(i)$ until one is found satisfying for $s_*=s_1$. In Figure 85, it is shown how in curve d the floor error can be avoided using this method.

The constellation of the interleaver used to obtain curve "d" is presented in Figure 86.

4.2.4- Simulations.

Simulations with:

(a) two equal, recursive convolutional consistent codes
(b) with 16 states,
(c) interleaver of length 4096 and 16384
(d) using S-random permutation with S=31 and S=40
(e) running each simulation at least 25 Mbits,

shows that the decoding algorithm converges down to BER=10^{-5} at E_b/N_o below 1 dB with nine iterations.

4.3. Serial Multiple Concatenated Convolutional Codes:

4.3.1 Encoder

An SMCCC Encoder comprises of two serial concatenated recursive systematic convolutional encoders separated by an interleaver. The encoders are arranged in a "serial concatenation". The concatenated recursive systematic convolutional encoders are identical.

Figure 87 represents the proposed encoder. A SMCCC encoder is a combination of two simple encoders. The input is a block of information bits. The two encoders generate parity symbols (u_0 and u'_0) from two simple recursive convolutional codes. The key innovation of this technique is an interleaver "I" , which permutes the original information bits before input to the second encoder. The permutation allows those input sequences for which one encoder produces low-weight codewords which will usually cause the other encoder to produce high-weight codewords. Thus, even though the constituent codes are individually weak, the combination is surprisingly powerful. The resulting code has features similar to a "random" block code.

In this way, we have the information symbols (u_1 and u_2) and two redundant symbols (u_0 and u'_0). With this redundancy it is possible to reach longer loops and to reduce the peak to average ratio (PAR) , at the cost of a slight increase of the constellation encoder .

In Figure 83 we present the conversion that we propose, taking into account the new parity bit.

4.3.2 Decoder

In Figure 88, the block diagram of an iterative decoder is shown. It is based on two modules denoted by "SISO" one for each encoder, an interleaver, and a deinterleaver. The SISO module is a four-port device, with two inputs and two outputs. It accepts as inputs the probability distributions of the information and code symbols labeling the edges of the code trellis, and forms as outputs an update of these distributions based upon the code constraints. The updated probabilities of the input and code symbols are used in the decoding procedure.

The SISO module is a four-port device that accepts at the input the sequences of probability distributions and outputs the sequences of probability distributions based on its inputs and on its knowledge of the code. The output probability distributions represent a smoothed version of the input distributions. The algorithm is completely general and capable of coping with parallel edges and also with encoders with rates greater than one, like those encountered in some concatenated schemes.

The SISO algorithm requires that the whole sequence has been received before starting the smoothing process. The reason is that backward recursion starts from the final trellis state. A more flexible decoding strategy is offered by modifying

the algorithm in such a way that the SISO module operates on a fixed memory span and outputs the smoothed probability distributions after a given delay, D. This new algorithm is called the sliding-window soft-input soft-output (SW-SISO) algorithm

The SW-SISO algorithm solves the problems of continuously updating the probability distributions, without requiring trellis terminations. Their computational complexity in some cases is around 5 times that of other suboptimal algorithms like SOVA. This is due mainly to the fact that they are ***multiplicative*** algorithms. In this section, we overcome this drawback by proposing the additive version of the SISO algorithm.

4.3.3. Interleaver design.

SMCCC does not have a problem with floor errors as does PMCCC. The floor error begins after 10^{-7} that made it suitable for ADSL applications. In an SMCCC the interleaver establishes a relationship between portions of a codeword. For a good SMCCC, we can design an interleaver of permutation length " p " that maximizes the minimum Hamming weight generated by weight two inputs. In an SMCCC, the interleaver establishes a relationship between portions of a code-word. In the SMCCC case, because one of the inputs come from the outer encoder, the roll of the interleaver is not so critical; for this reason the method proposed for the interleaver is to disperse symbols as widely as possible in a "constellation way". One effective method is to choose for each $i \in [1,p]$ $\pi(i)=p/3*i$. An example of this method is show in Figure 89.

4.3.4- Simulations.

Simulations with two equal, recursive convolutional consistent codes with 16 states and an interleaver of length between 100 and 1000 using S-random permutation, and each simulation run examined at least 25 Mbits show that the decoding algorithm converges down to BER=10^{-7} at E_b/N_o of below 1 dB with less than nine iterations.

4.4 The number of iterations in the decoder.

The number of iterations is a very important subject for the different applications of PMCCC and SMCCC. For applications where the delay is not important, a large number is acceptable. For real time applications or for quasi-real time applications it is important to use a number of iterations as low as possible maintaining the advantages of this technique. The necessary number of iterations depends upon the E_b/N_o ratio in the receiver. In Figure 51, we present this relationship for the SMCCC case, we represent values of E_b/N_o below 0.1 dB, for values around 2 dB it is sufficient to use a number below 10 iterations.

4. 5.Comparisons

The PMCCC has a floor-error around a BER of 10^{-6}. The reason for this is that the SMCCC functions in an inner and outer encoder structure, while the PMCCC functions as two parallel encoders. In Figure 52 we present the floor error effect for PMCCC and that SMCCC does not show the floor error effect at least until BER of 10^{-9}. For simulation after 10^{-9} a lot of time is required and it is not possible to give a simulation result.

5. Reed-Solomon Codes and Turbo Codes for ADSL systems

5.1 Encoder

Figure 90 represents the proposed encoder. A PMCCC encoder is a combination of two simple encoders. The input is a block of information bits. The two encoders generate parity symbols (u_0 and u'_0) from two simple recursive convolutional codes. The key innovation of this technique is an interleaver "I", which permutes the original information bits before input to the second encoder. The permutation performed by the interleaver allows those input sequences for which one encoder produces low-weight codewords to usually cause the other encoder to produce high-weight codewords. Thus, even though the constituent codes are individually weak, the combination is surprisingly powerful. The resulting code has features similar to a "random" block code.

In this way, we have the information symbol (u_1) and two redundant symbols (u_0 and u'_0). With this redundancy it is possible to reach longer loops, or works at higher bit rates in the same loop, at the cost of a slight increase of the constellation encoder.

In a preferred embodiment, the reason we suggest the use two 1/2 convolutional encoders in parallel, is that this simplifies the Turbo-code encoder and it still produces results very close to the channel capacity. The use of two 2/3 convolutional encoders in parallel will produce a little better result (in the order of 0.1 or 0.2 dB) and will increase the complexity at least by four. From a practical point of view we think that is not necessary.

In Figure 83, we have presented the conversion that we propose, taking into account the new parity bit.

5.2 Decoder.

In Figure 84, we present the decoder that uses an iterative technique, using two soft-decision input/output trellis decoders in each decoding state. The Maximum-a-Posteriori (MAP) Trellis decoder provides the soft output result suitable for turbo-code decoding.

5.3. Simulations.

In Figure 91, we present the simulation results that we obtained, with two equal, recursive convolutional consistent codes with an interleaver of length 400 and *K=50*. With this value the delay is below 5 msec for 1.5 Mbit/s assuming a delay of 3 interleaver.

The number of Iterations is always below 10.

The convolutional encoder used is presented in Figure 92.

6. Forward Error Correction with Low-density parity-check codes

Low-density parity-check codes are codes specified by a matrix containing mostly 0's and only a small number of 1's. In particular, an (n, j, k) low-density code is a code of block length n with a matrix like that of Table 24 where each column contains a small fixed number, j, of 1's and each row contains a small fixed number, k, of 1' s. Note that this type of matrix does not have the check digits appearing in diagonal form as in Table 25. However, for coding purposes, the equations represented by these matrices can always be solved to give the check digits as explicit sums of information digits.

Table 24 Example of a low-density code matrix; N = 20, j = 3, k = 4.

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1

Tables 25. Example of parity-check matrix.

INFORMATION DIGITS				CHECK DIGITS		
x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	1	1	0	1	0	0
1	1	0	1	0	1	0
1	0	1	1	0	0	1

$\longleftrightarrow$

$$x_5 = x_1 \oplus x_2 \oplus x_3$$
$$x_6 = x_1 \oplus x_2 \oplus x_4$$
$$x_7 = x_1 \oplus x_3 \oplus x_4$$

These codes are not optimum in the somewhat artificial sense of minimizing probability of decoding error for a given block length, and it can be shown that the maximum rate at which these codes can be used is bounded below channel capacity. However, a very simple decoding scheme exists for low-density codes, and this compensates for their lack of optimality.

The analysis of a low-density code of long block length is difficult because of the immense number of code words involved. It is simpler to analyze a whole ensemble of such codes because the statistics of an ensemble permit one to average over quantities that are not tractable in individual codes. From the ensemble behavior, one can make statistical statements about the properties of the member codes. Furthermore, one can with high probability find a code with these properties by random selection from the ensemble.

In order to define an ensemble of (n, j, k) low-density codes, consider Table 24 again. Note that the matrix is divided into j sub-matrices, each containing a single 1 in each column. The first of these sub-matrices contains all its 1's in descending order, i.e., the i' th row contains 1's in columns $(i - 1)k + 1$ to ik. The other sub-matrices are merely column permutations of the first. We define an ensemble of (n, j, k) codes as the ensemble resulting from random permutation of the columns of each of the bottom $j - 1$ sub-matrices of a matrix such as Table 24, with equal probability assigned to each permutation. There are two interesting results that can be proven using this ensemble, the first concerning the minimum distance of the member codes, and the second concerning the probability of decoding error.

The minimum distance of a code is the number of positions in which the two nearest code words differ. Over the ensemble, the minimum distance of a member code is a random variable, and it can be shown that the distribution function of this random variable can be over bounded by a function. As the block length increases, for fixed $j \geq 3$ and $k > j$, this function approaches a unit step at a fixed fraction δ_{ik} of the block length. Thus, for large n, practically all the codes in the ensemble have a minimum distance of at least $n\delta_{ik}$,. In Table 26 this ratio of typical minimum distance to block length is compared to that for a parity-check code chosen at random, i.e., with a matrix filled in with equiprobable independent binary digits. It should be noted that for all the specific nonrandom procedures known for constructing codes, the ratio of minimum distance to block length appears to approach 0 with increasing block length.

The probability of error using maximum likelihood decoding for low-density codes clearly depends upon the particular channel on which the code is being used. The results are particularly simple for the case of the BSC, or binary symmetric channel, which is a binary-input, binary-output, memoryless channel with a fixed probability of transition from either input to the opposite output. Here it can be shown that over a reasonable range of channel transition probabilities, the low-density code has a probability of decoding error that decreases exponentially with block length and that the exponent is the same as that for the optimum code of slightly higher rate as given in Table 27 .

Table 26. Comparison of δ_{jk}, the ratio of typical minimum distance to block length for an (n, j, k) code, to δ, the same ratio for an ordinary parity-check code of the same rate.

j	k	Rate	δ_{jk}	δ
5	6	0.167	0.255	0.263
4	5	0.2	0.210	0.241
3	4	0.25	0.122	0.214
4	6	0.333	0.129	0.173
3	5	0.4	0.044	0.145
3	6	0.5	0.023	0.11

Table 27. Loss of rate associated with low-density codes.

j	k	Rate	RATE FOR EQUIVALENT OPTIMUM CODE
3	6	0.5	0.555
3	5	0.4	0.43
4	6	0.333	0.343
3	4	0.25	0.266

Although this result for the BSC shows how closely low-density codes approach the optimum, the codes are not designed primarily for use on this channel. The BSC is an approximation to physical channels only when there is a receiver that makes decisions on the incoming signal on a bit-to-bit basis. Since the decoding procedure to be described later can actually use the channel a posteriori pryobabilities, and since a bit-by-bit decision throws away available information, we are actually interested in the probability of decoding error of a binary-input, continuous-output channel. If the noise affects the input symbols symmetrically, then this probability can again be bounded by an exponentially decreasing function of the block length, but the exponent is a rather complicated function of the channel and code. It is expected that the same type of result holds for a wide class of channels with memory, but no analytical results have yet been derived. For channels with memory, it is clearly advisable, however, to modify the ensemble somewhat, particularly by permuting the first sub-matrix and possibly by changing the probability measure on the permutations.

7. Application to Modem Communications Systems

In a preferred embodiment, the use of PMCCC or SMCCC is negotiated independently in each direction of communication in the system. In the case of ADSL, this permits the use of trellis code in one direction and a SMCCC code in the other direction.

Computer Program Listing

This patent application includes a computer program listing containing 37 pages, and included as an appendix. The program relates to a Reed-Solomon Encoder and Decoder and a PMCCC Encoder and Decoder for 2 parallel concatenated convolutional codes.

Thus it is seen that the objects, features and advantages of the present invention are efficiently obtained. The preferred embodiment described herein is intended to disclose the best mode of the invention and to teach those having ordinary skill in the art how to make and use the invention, but should not be interpreted as limiting the scope and spirit of the invention as embodied in the appended claims.

What We Claim Is:

1. A method of forward error correction for communication systems, comprising:

producing a symbol stream by forward error coding of a data stream using multiple concatenated coders connected by an interleaver, all of the concatenated coders being of a single type that is one of convolutional and non-convolutional;

modulating said symbol stream to produce a modulated signal; and,

transmitting said modulated signal over a communication link.

2. The method recited in Claim 1 wherein said communication system is a wired system.

3. The method recited in Claim 1 wherein said communication system is an optical system.

4. The method recited in Claim 1 wherein the modulating is accomplished with a multicarrier method.

5. The method recited in Claim 4 wherein the multicarrier method is a Discrete Multi-Tone (DMT) method.

6. The method recited in Claim 1 wherein the modulating is accomplished with a CAP-QAM single carrier method.

7. The method recited in Claim 1 wherein the modulating is accomplished with a Quadrature Amplitude Modulation (QAM) method.

8. The method recited in Claim 1 wherein the modulating is accomplished using a Pulse Amplitude Modulation (PAM) method.

9. The method recited in Claim 1 wherein said multiple concatenated coders are convolutional coders.

10. The method recited in Claim 1 wherein a first of the concatenated coders is configured in parallel with the interleaver and a second of the concatenated coders.

11. The method recited in Claim 1 wherein a first of the concatenated coders is configured in series with the interleaver and a second of the concatenated coders.

12. The method recited in Claim 1 wherein said multiple concatenated coders are non-convolutional coders.

13. The method recited in Claim 12 wherein one of said non-convolutional coders comprises a Reed-Solomon encoder.

14. The method recited in Claim 12 wherein one of said non-convolutional coders comprises a low density parity check encoder.

16. A method of forward error correction for communication systems, comprising:

producing a symbol stream by forward error coding of a data stream using an outer level non-convolutional coder and an inner level coder comprising multiple concatenated coders connected by an interleaver, all of the concatenated coders being of a single type that is one of convolutional and non-convolutional;

modulating said symbol stream to produce a modulated signal; and,

transmitting said modulated signal over a communication link.

17. The method recited in Claim 16 wherein said outer level coder comprises a Reed-Solomon Encoder.

18. A method of peak power level reduction for communication systems utilizing a plurality of coders comprising the following steps:

producing a peak reduced signal by encoding said data stream by said plurality of coders;

modulating said peak reduced signal; and,

transmitting the modulated peak reduced signal.

19. A method of forward error correction for communication systems, comprising the following steps:

producing a symbol stream by forward error coding of a data stream using multiple concatenated coders connected by an interleaver, all of the concatenated coders being of a single type that is one of convolutional and non-convolutional;

modulating said symbol stream to produce a modulated signal;

transmitting said modulated signal over a communication link;

receiving said modulated signal, where said received modulated signal includes errors;

demodulating said received signal which includes errors;

iteratively decoding said demodulated signal using multiple concatenated decoders connected by an interleaver and a de-interleaver, all of the concatenated decoders being of the same type as the concatenated coders; and,

regenerating said data stream and eliminating said errors.

20. A method of forward error correction for communication systems, comprising:

receiving a modulated signal from a communications link, where said received modulated signal includes errors;

demodulating said received signal which includes errors;

iteratively decoding said demodulated signal using multiple concatenated decoders connected by an interleaver and a de-interleaver, all of the concatenated decoders being of a single type that is one of convolutional and non-convolutional; and,

regenerating said data stream and eliminating said errors.

21. An apparatus for forward error correction for communication systems, comprising:

multiple concatenated coders connected by an interleaver for producing a symbol stream by forward error coding of a data stream, all of the concatenated coders being of a single type that is one of convolutional and non-convolutional,;

a modulator for modulating said symbol stream to produce a modulated signal; and,

a transmitter for transmitting said modulated signal over a communication link.

22. The apparatus recited in Claim 21 wherein said communication system is a wired system.

23. The apparatus recited in Claim 21 wherein said communication system is an optical system.

24. The apparatus recited in Claim 21 wherein said multiple concatenated coders are convolutional coders.

25. The apparatus recited in Claim 21 wherein a first of the concatenated coders is configured in parallel with the interleaver and a second of the concatenated coders.

26. The apparatus recited in Claim 21 wherein a first of the concatenated coders is configured in series with the interleaver and a second of the concatenated coders.

27. The apparatus recited in Claim 21 wherein said multiple concatenated coders are non-convolutional coders.

29. An apparatus for forward error correction for communication systems, comprising:

an outer level non-convolutional coder and an inner level coder comprising multiple concatenated coders connected by an interleaver for producing a symbol stream by forward error coding of a data stream, all of the concatenated coders being of a single type that is one of convolutional and non-convolutional;

a modulator for modulating said symbol stream to produce a modulated signal; and,

a transmitter for transmitting said modulated signal over a communication link.

30. The apparatus recited in Claim 29 wherein said outer level coder comprises a Reed-Solomon encoder.

31. An apparatus for accomplishing peak power level reduction for communication systems utilizing a plurality of coders comprising:

means for producing a peak reduced signal by encoding said data stream by said plurality of coders;

means for modulating said peak reduced signal; and,

means for transmitting the modulated peak reduced signal.

32. An apparatus for forward error correction for communication systems, comprising:

a receiver for receiving a modulated signal from a communications link, where said received modulated signal includes errors;

a demodulator for demodulating said received signal which includes errors;

multiple concatenated decoders connected by an interleaver and a de-interleaver for iteratively decoding said demodulated signal, all of the concatenated decoders being of a single type that is one of convolutional and non-convolutional; and,

means for regenerating said data stream and eliminating said errors.

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : H04L 1/00 — A1

(11) International Publication Number: **WO 00/07323**

(43) International Publication Date: 10 February 2000 (10.02.00)

(21) International Application Number: PCT/US99/17369

(22) International Filing Date: 30 July 1999 (30.07.99)

(30) Priority Data:

60/094,629	30 July 1998 (30.07.98)	US
60/098,394	30 August 1998 (30.08.98)	US
60/133,390	10 May 1999 (10.05.99)	US

(71) Applicant *(for all designated States except US)*: VOCAL TECHNOLOGIES, LTD. [US/US]; 200 John James Audubon Parkway, Buffalo, NY 14228 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: TORRES, Juan Alberto [ES/US]; 13 Quaker Lake Terrace, Orchard Park, NY 14127 (US). DEMJANENKO, Victor [US/US]; 4420 Beach Ridge Road, Pendleton, NY 14094 (US). HIRZEL, Frederic [US/US]; 15754 Pinto Court, Clinton Township, MI 48035 (US).

(74) Agent: SIMPSON, Robert, P.; Simpson, Simpson & Snyder, HSBC Bank Building, Suite 200, 5554 Main Street, Williamsville, NY 14221 (US).

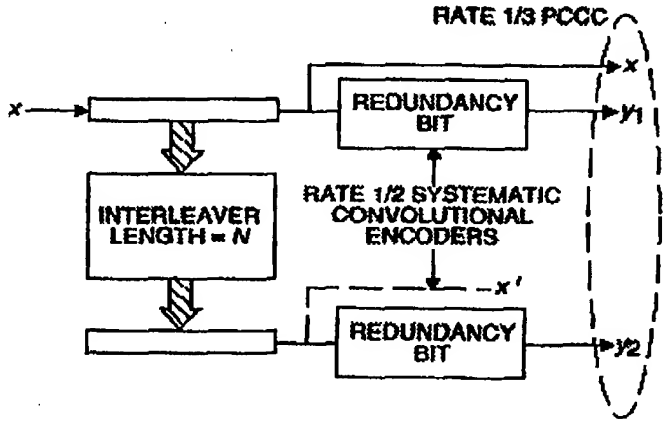
(81) Designated States: JP, US, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: FORWARD ERROR CORRECTING SYSTEM WITH ENCODERS CONFIGURED IN PARALLEL AND/OR SERIES

(57) Abstract

A method of forward error correction for communication systems, comprising the steps of producing a symbol stream by forward error coding of a data stream, modulating the symbol stream to produce a modulated signal, transmitting the modulated signal over a communication link, receiving the modulated signal, where the received modulated signal includes errors, demodulating the received signal which includes errors, decoding the demodulated signal by a plurality of convolutional decoders, and, regenerating the data stream and eliminating the errors. A method of peak power level reduction for communication systems utilizing a plurality of coders comprising the steps of producing a peak reduced signal by encoding the data stream by the plurality of coders, modulating the peak reduced signal, and, transmitting the modulated peak reduced signal. An apparatus is described for implementing the method.



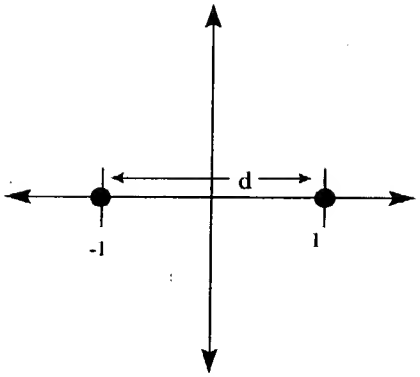


Figure 1

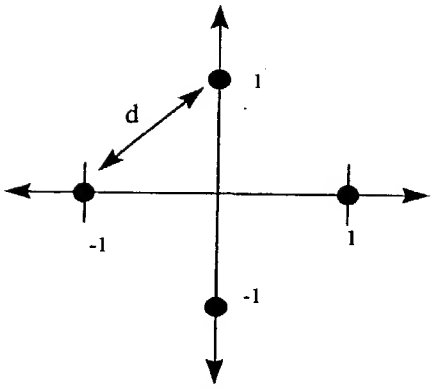


Figure 2

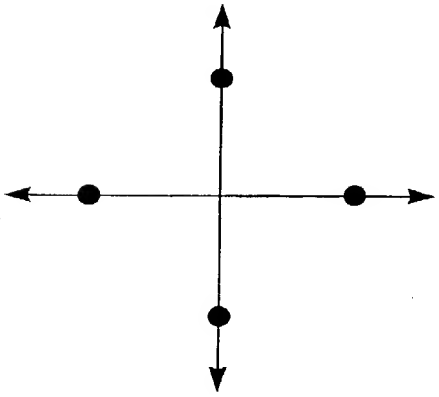


Figure 3

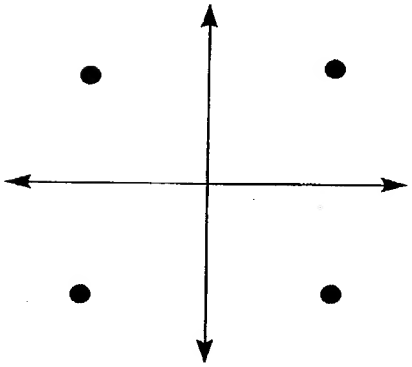


Figure 4

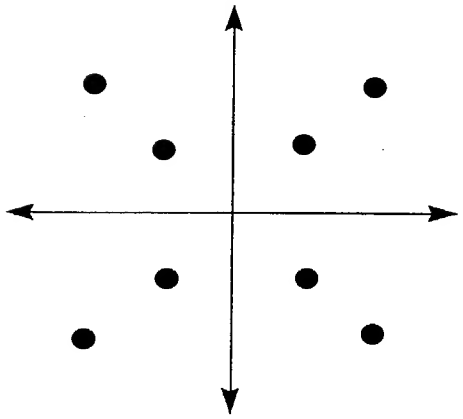


Figure 5

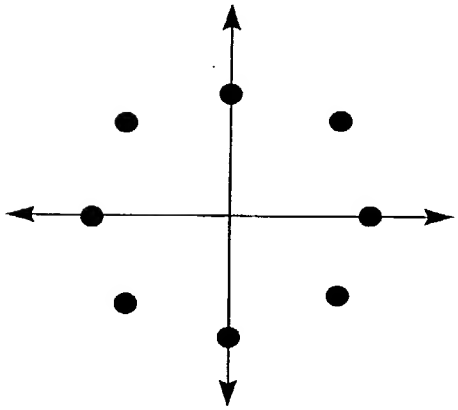


Figure 6

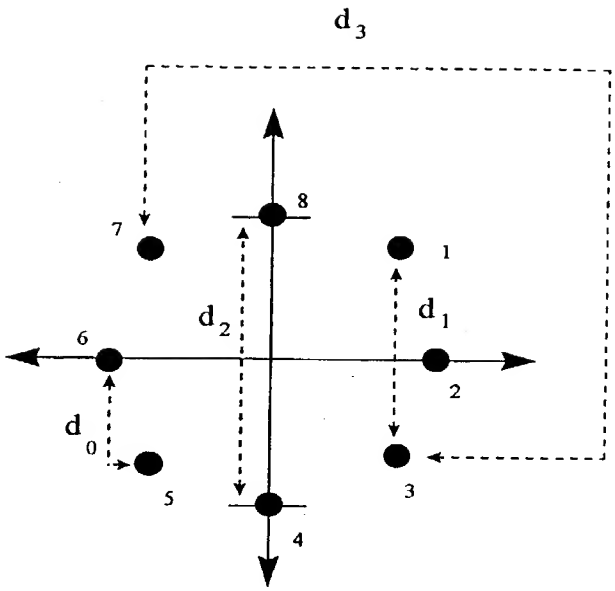


Figure 7

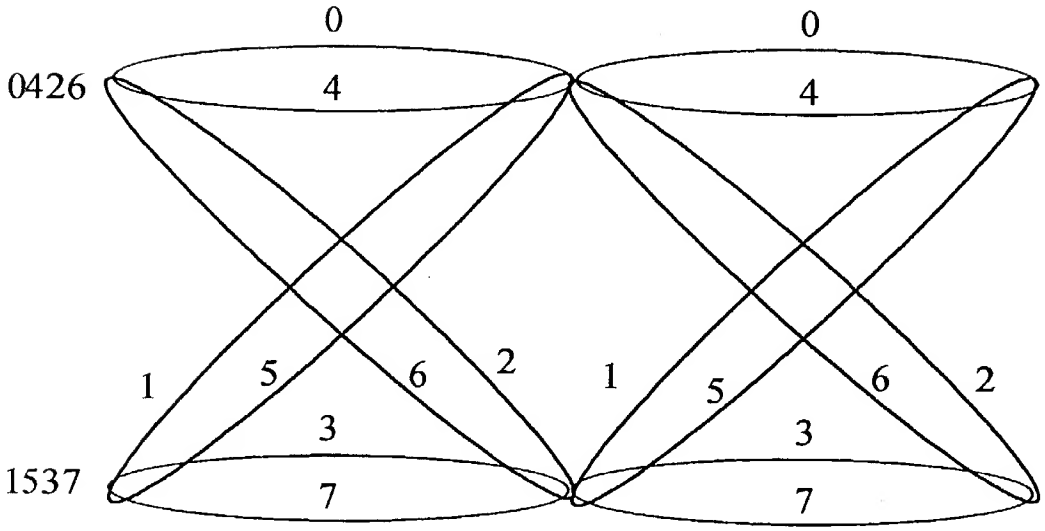


Figure 8

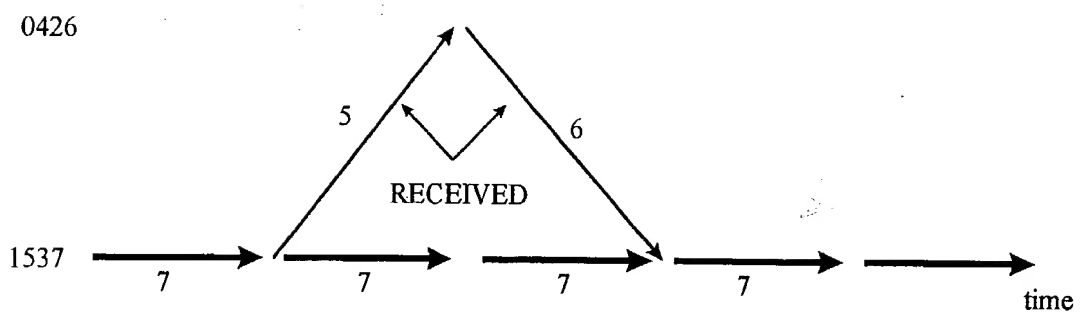


Figure 9

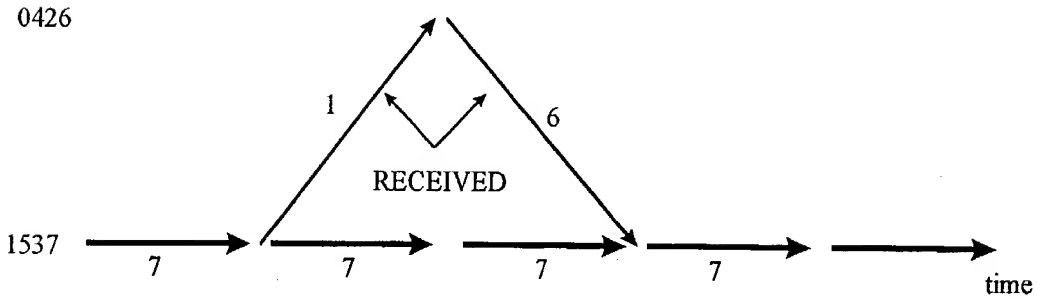


Figure 10

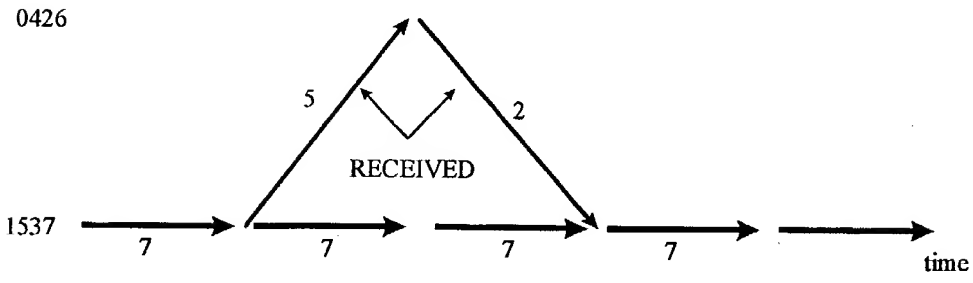


Figure 11

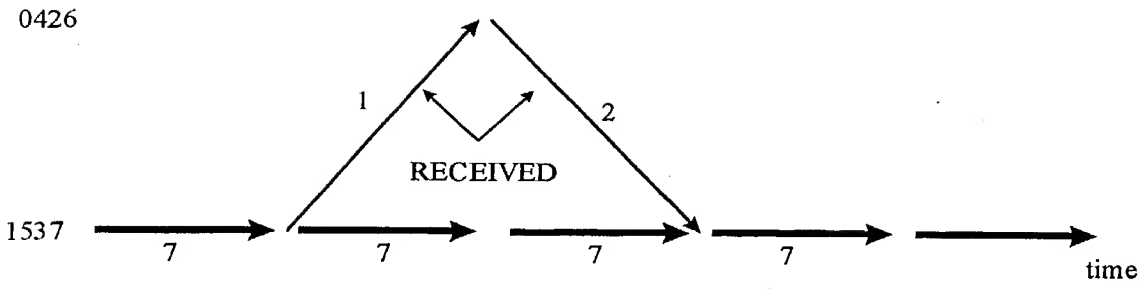


Figure 12

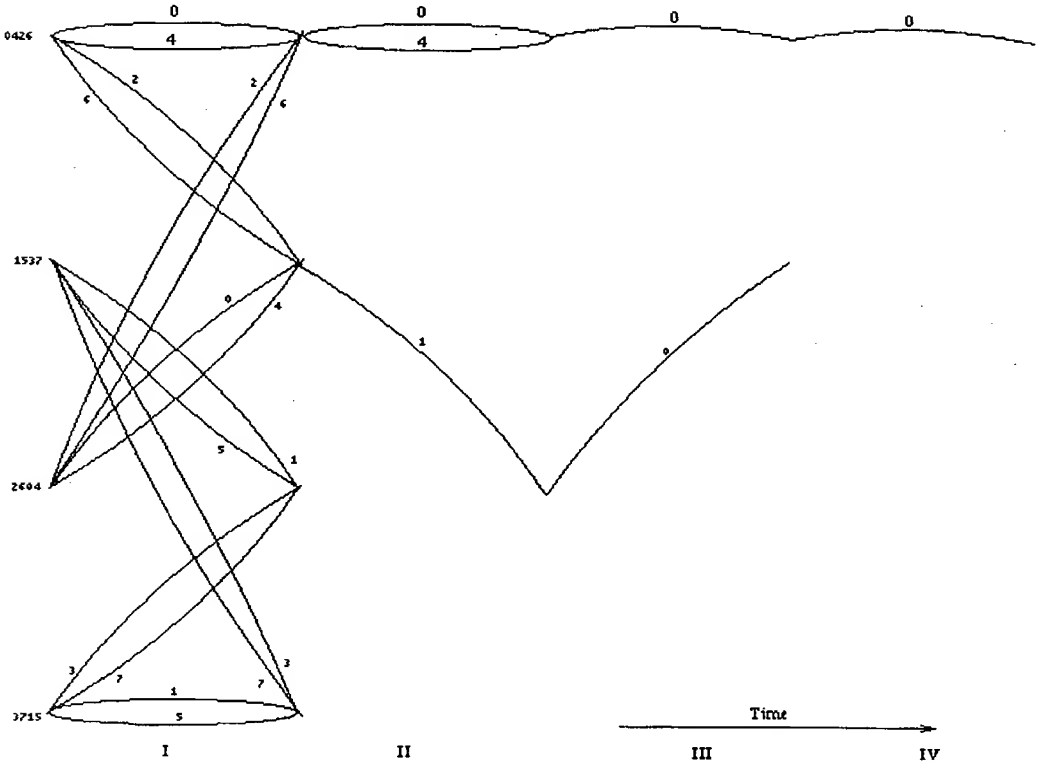


Figure 13

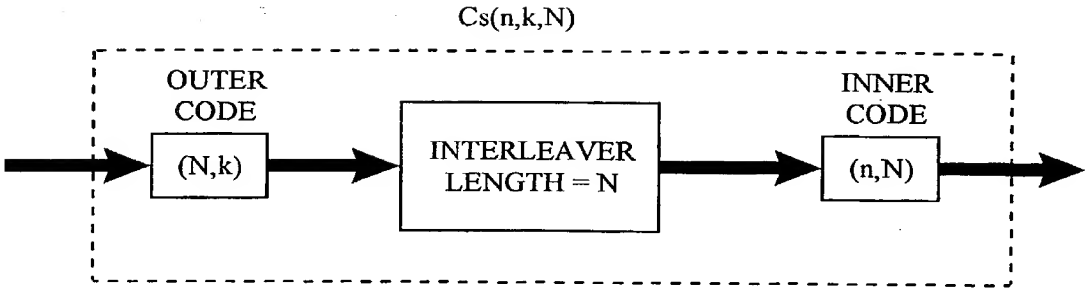


Figure 14

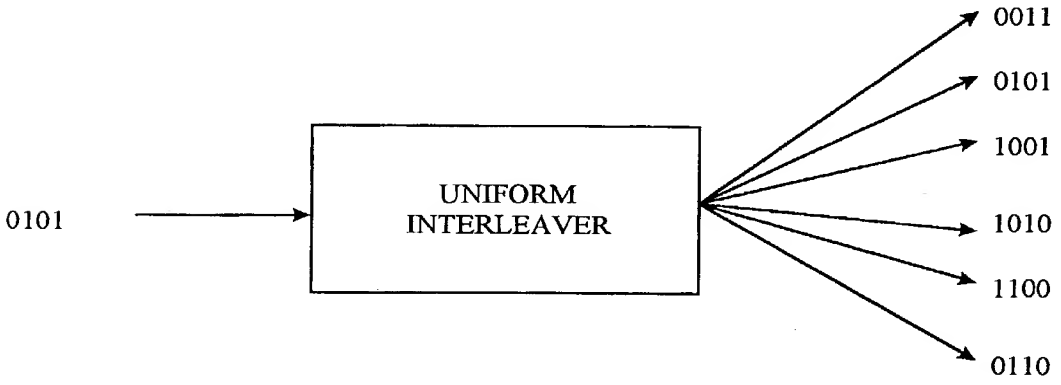


Figure 15

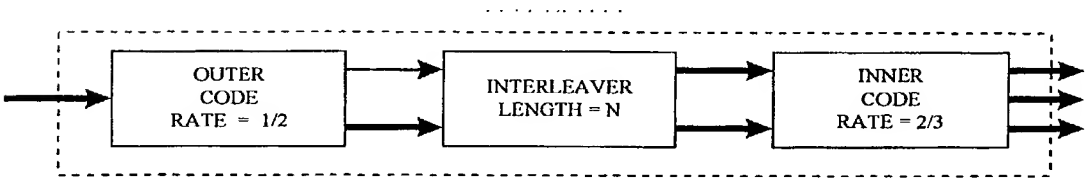


Figure 16

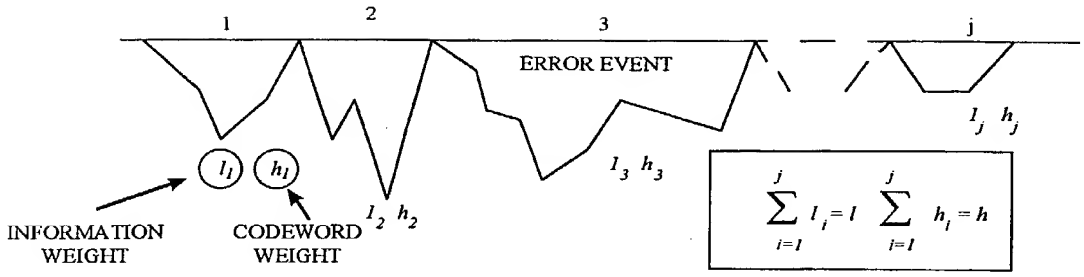


Figure 17

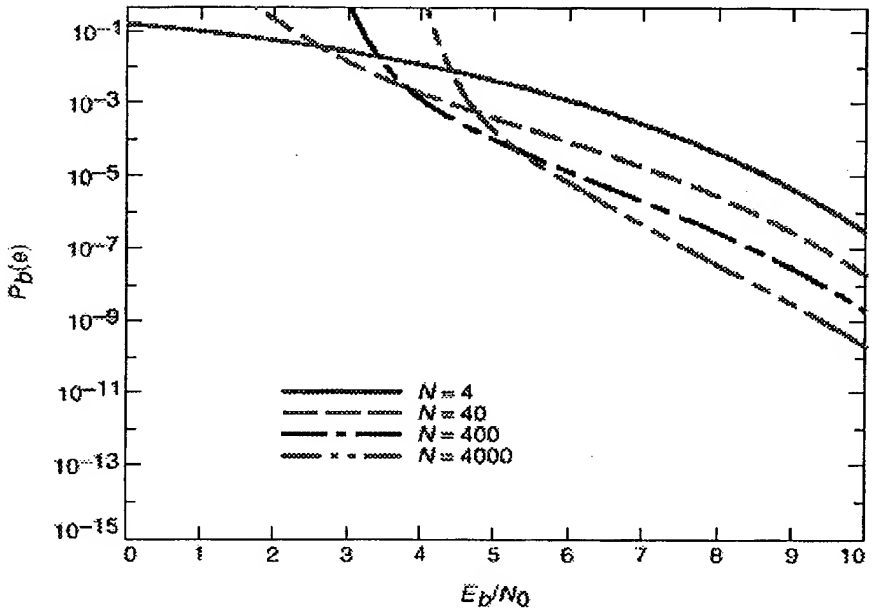


Figure 18

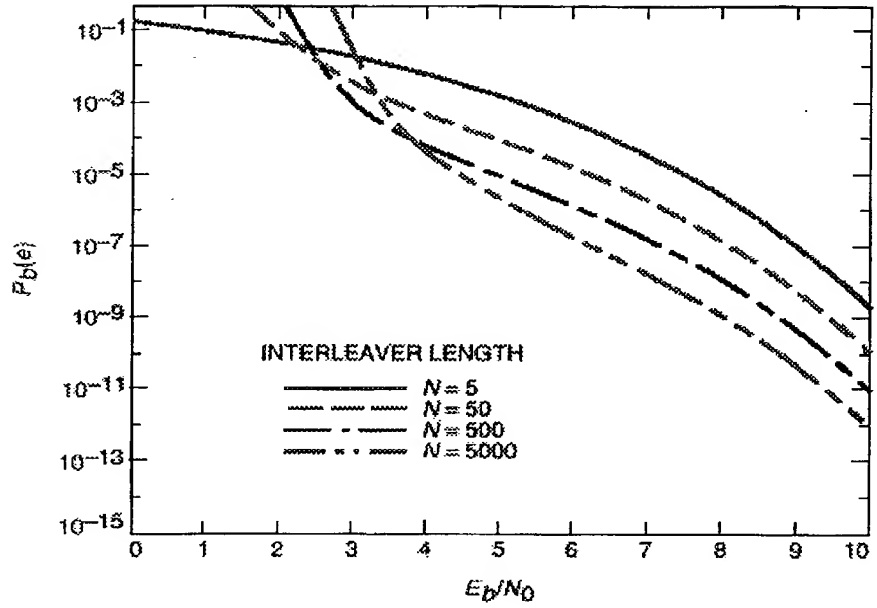


Figure 19

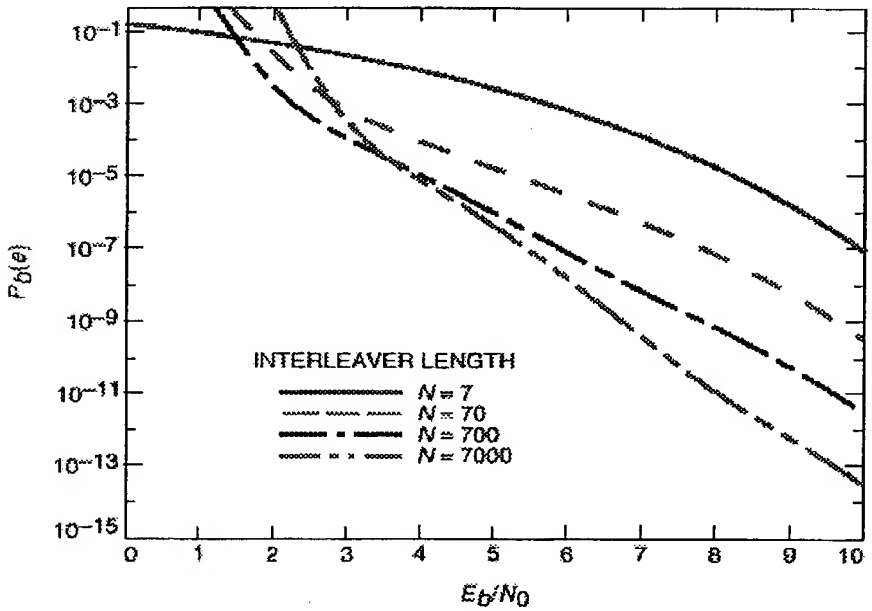


Figure 20

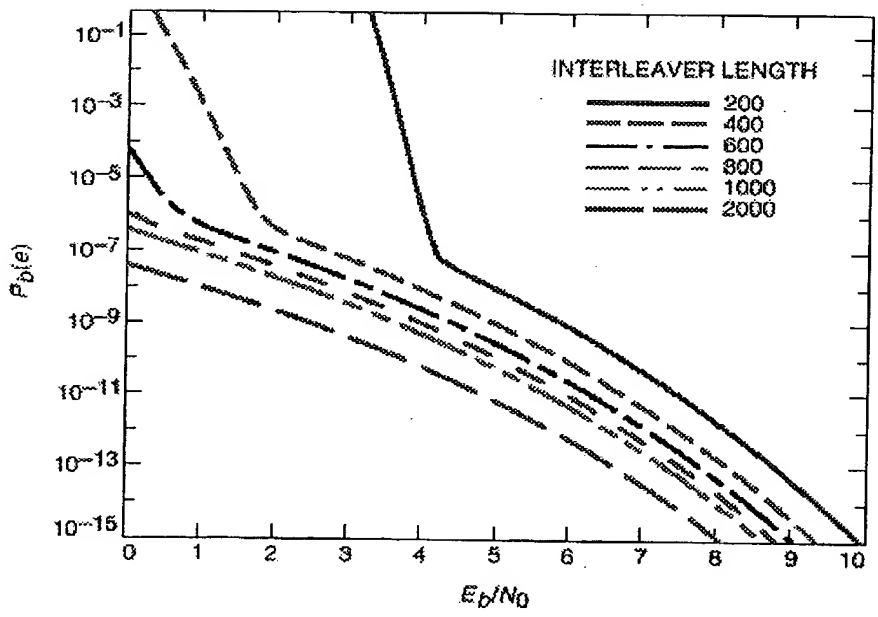


Figure 21

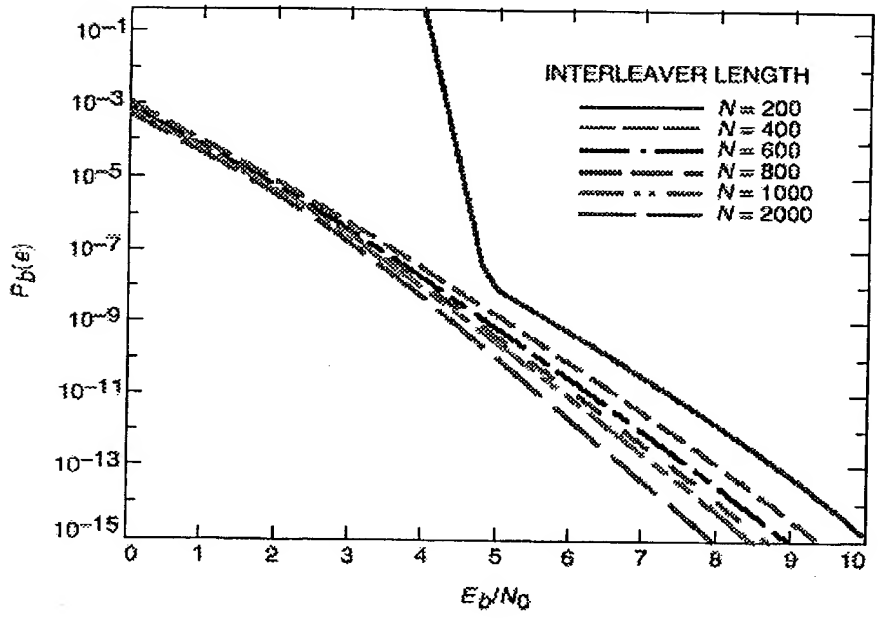


Figure 22

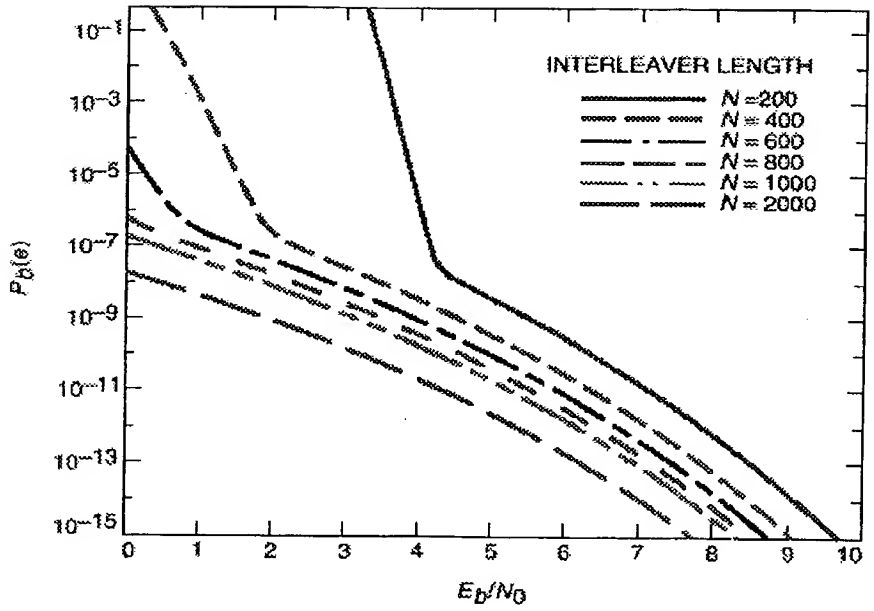


Figure 23

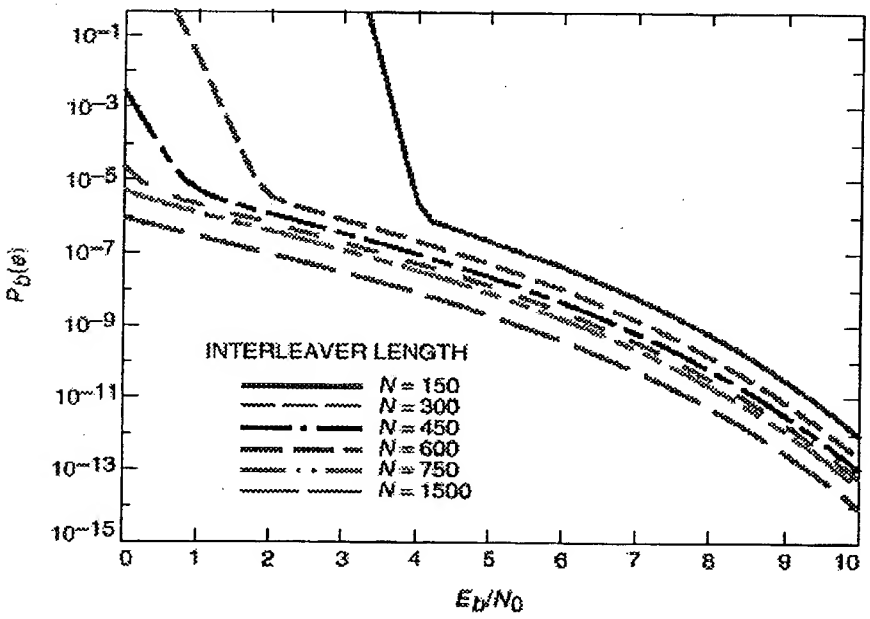


Figure 24

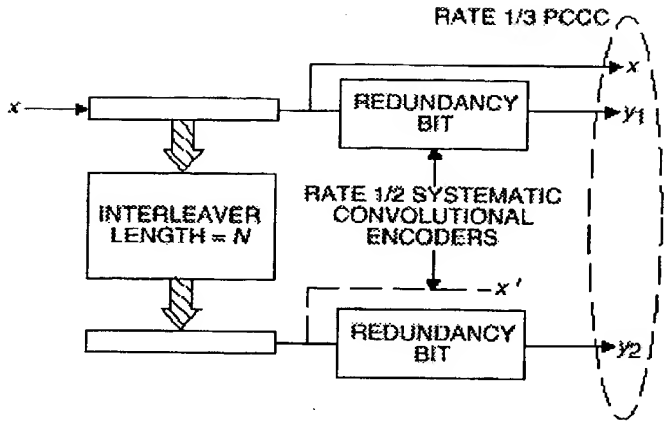


Figure 25

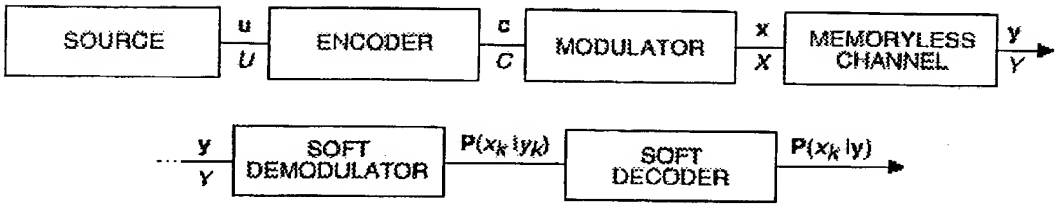


Figure 26

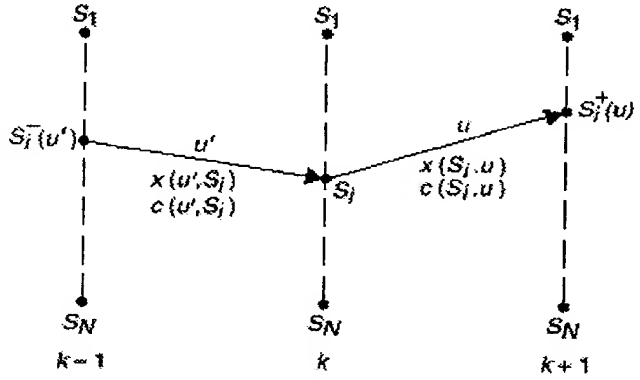


Figure 27

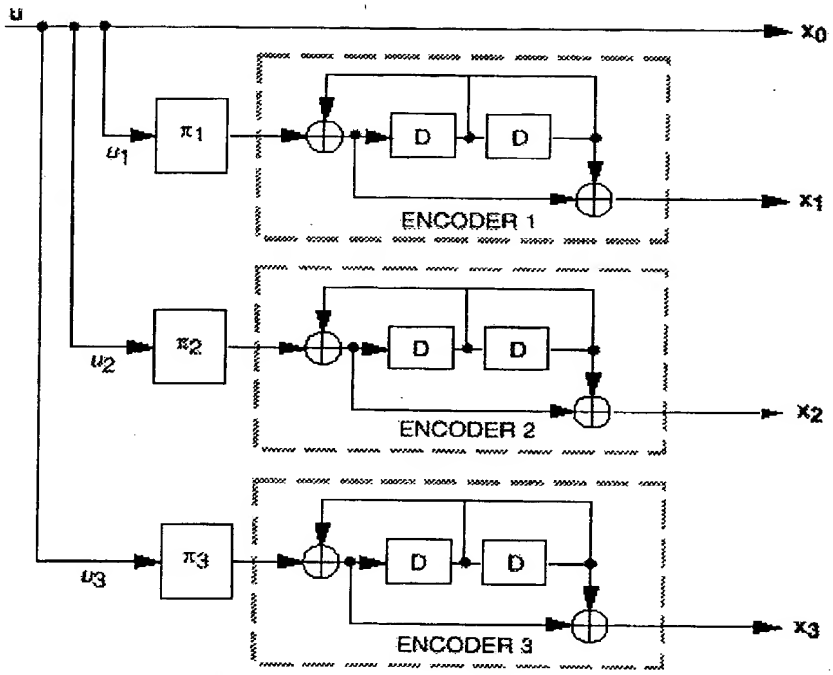


Figure 28

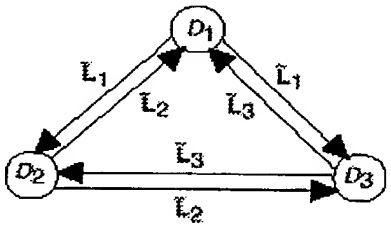


Figure 29

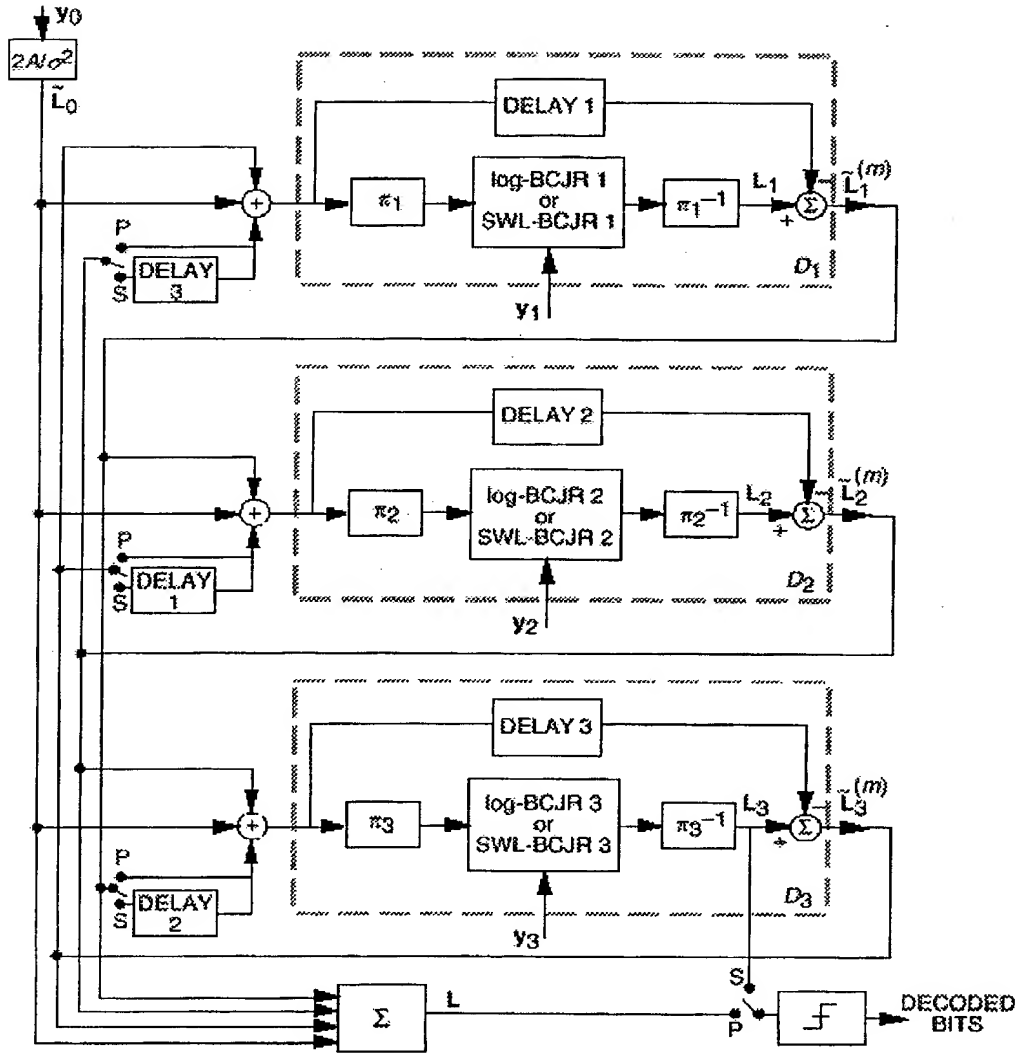

y0
2A/σ²
L̃0
DELAY 1
π1
log-BCJR 1 or SWL-BCJR 1
π1⁻¹
L1
L̃1(m)
D1
y1
P
S
DELAY 3
DELAY 2
π2
log-BCJR 2 or SWL-BCJR 2
π2⁻¹
L2
L̃2(m)
D2
y2
DELAY 1
DELAY 3
π3
log-BCJR 3 or SWL-BCJR 3
π3⁻¹
L3
L̃3(m)
D3
y3
DELAY 2
Σ
L
DECODED BITS

Figure 30

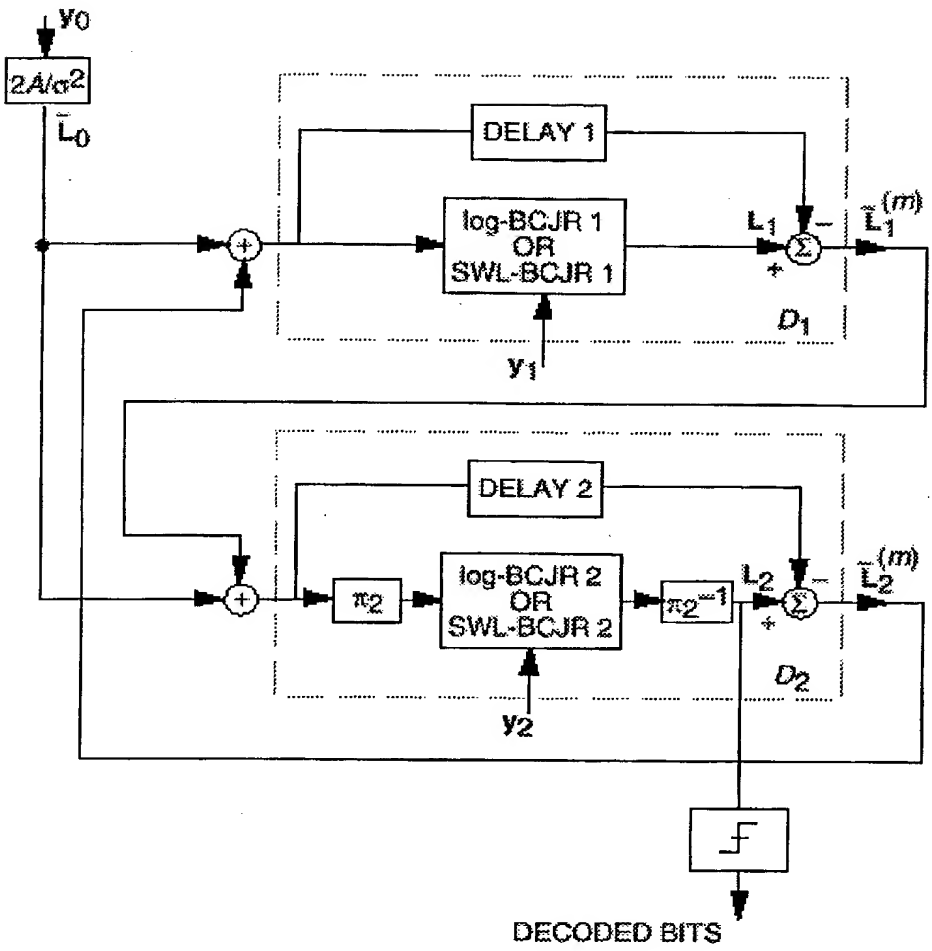

y0
2A/σ2
L0
DELAY 1
log-BCJR 1
OR
SWL-BCJR 1
L1
D1
y1
DELAY 2
π2
log-BCJR 2
OR
SWL-BCJR 2
π2-1
L2
D2
y2
DECODED BITS

Figure 31

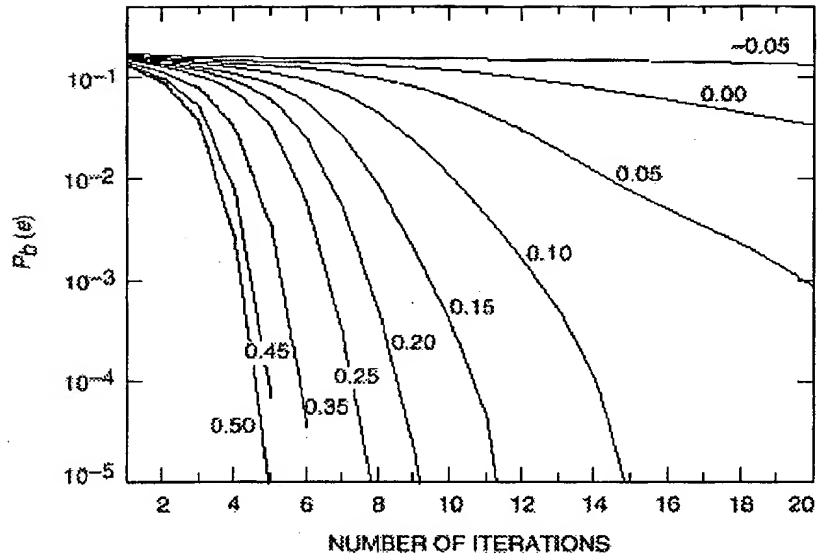


Figure 32

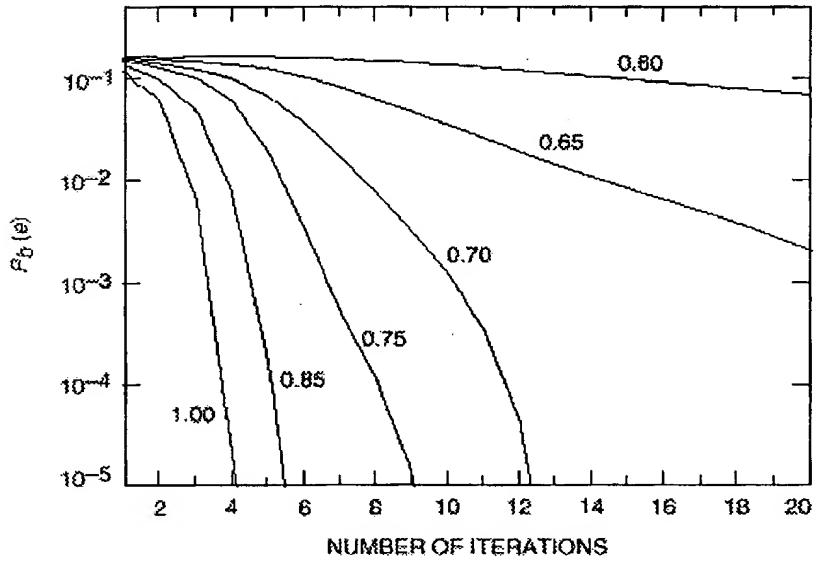


Figure 33

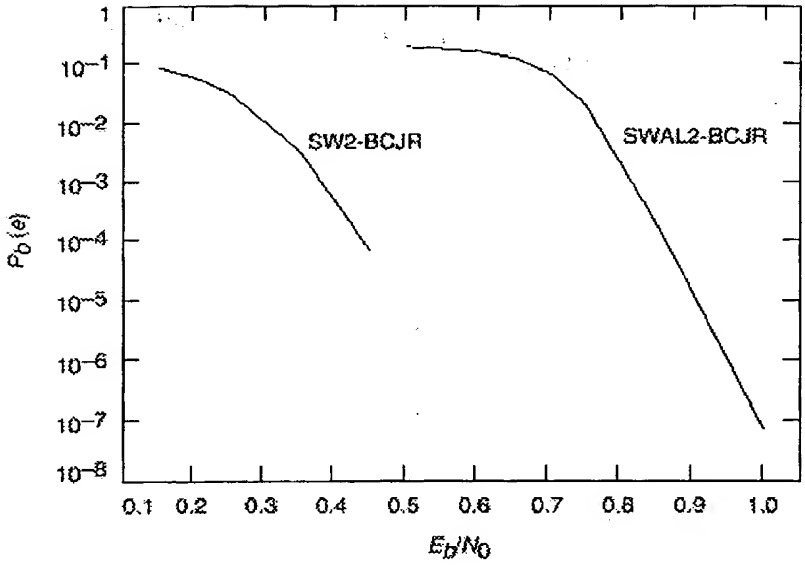


Figure 34

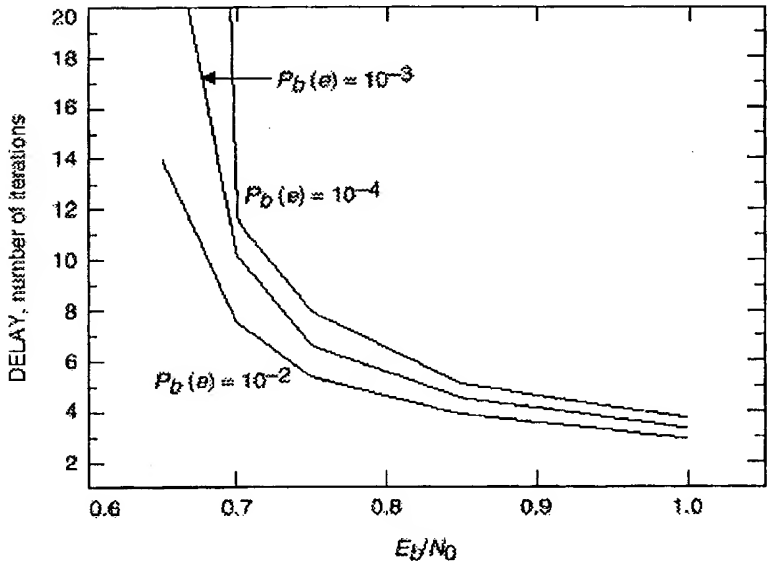


Figure 35

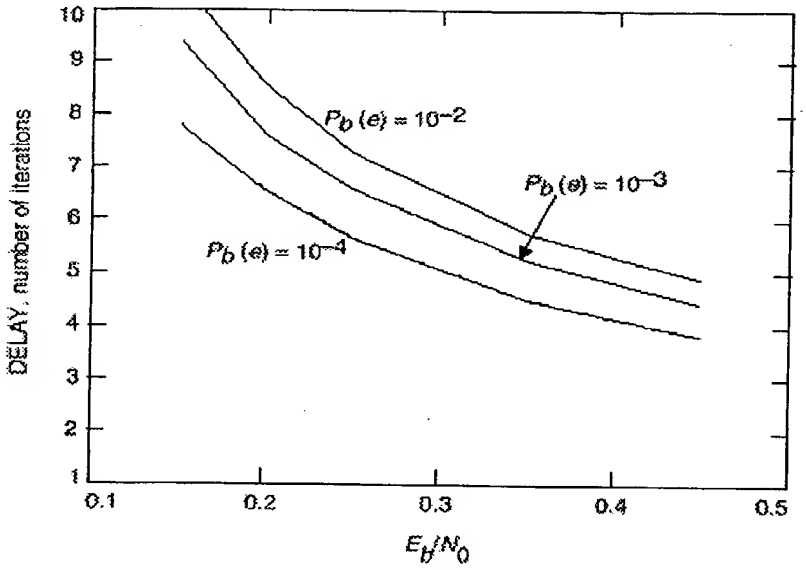

DELAY, number of iterations
10
9
8
7
6
5
4
3
2
1
0.1
0.2
0.3
0.4
0.5
E_b/N_0
$P_b(e) = 10^{-2}$
$P_b(e) = 10^{-3}$
$P_b(e) = 10^{-4}$

Figure 36

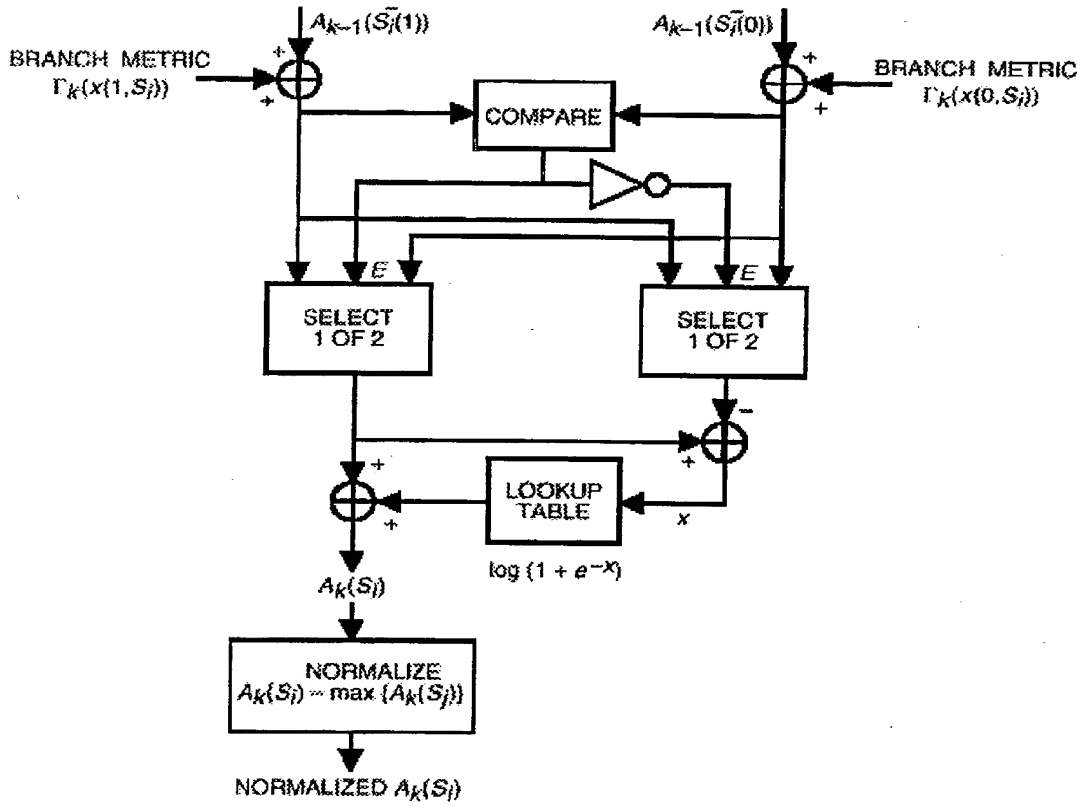


Figure 37

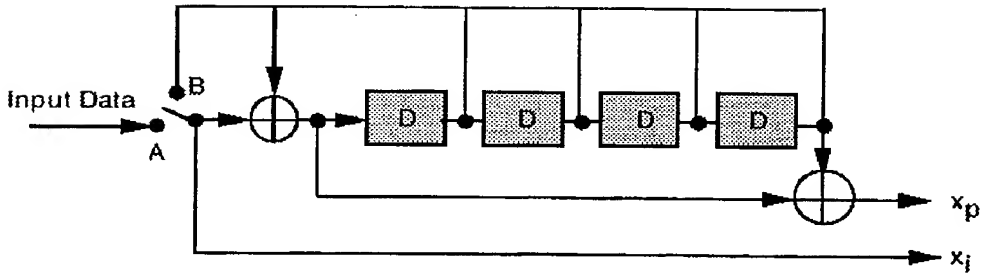


Figure 38

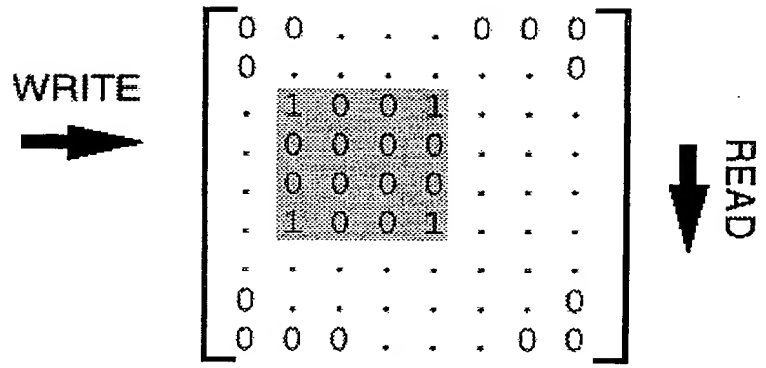


Figure 39

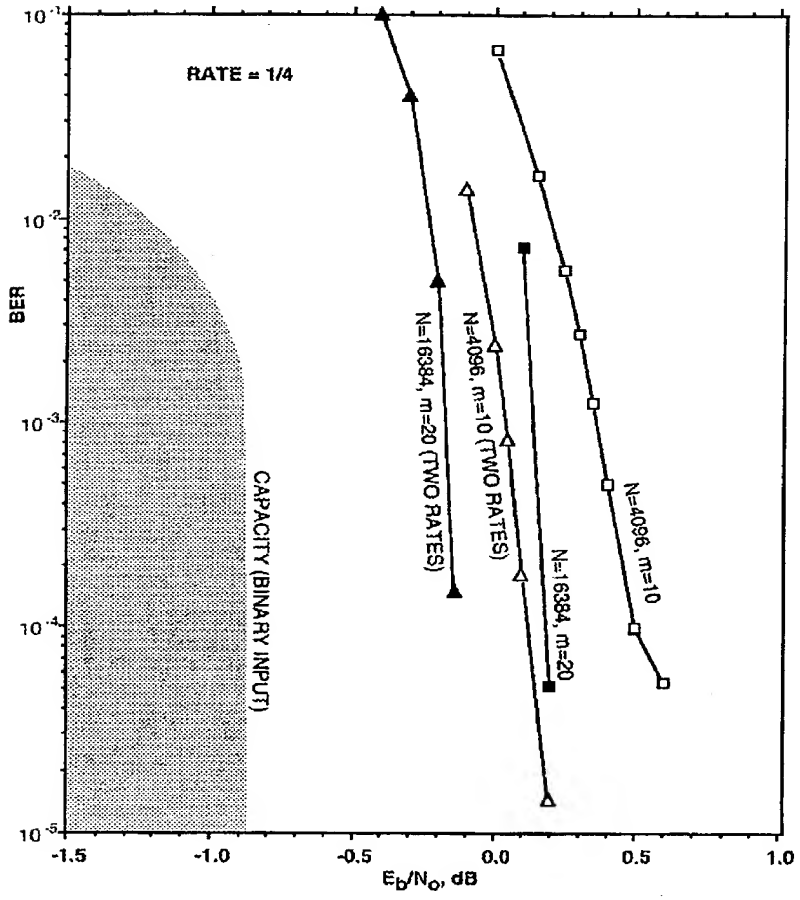


Figure 40

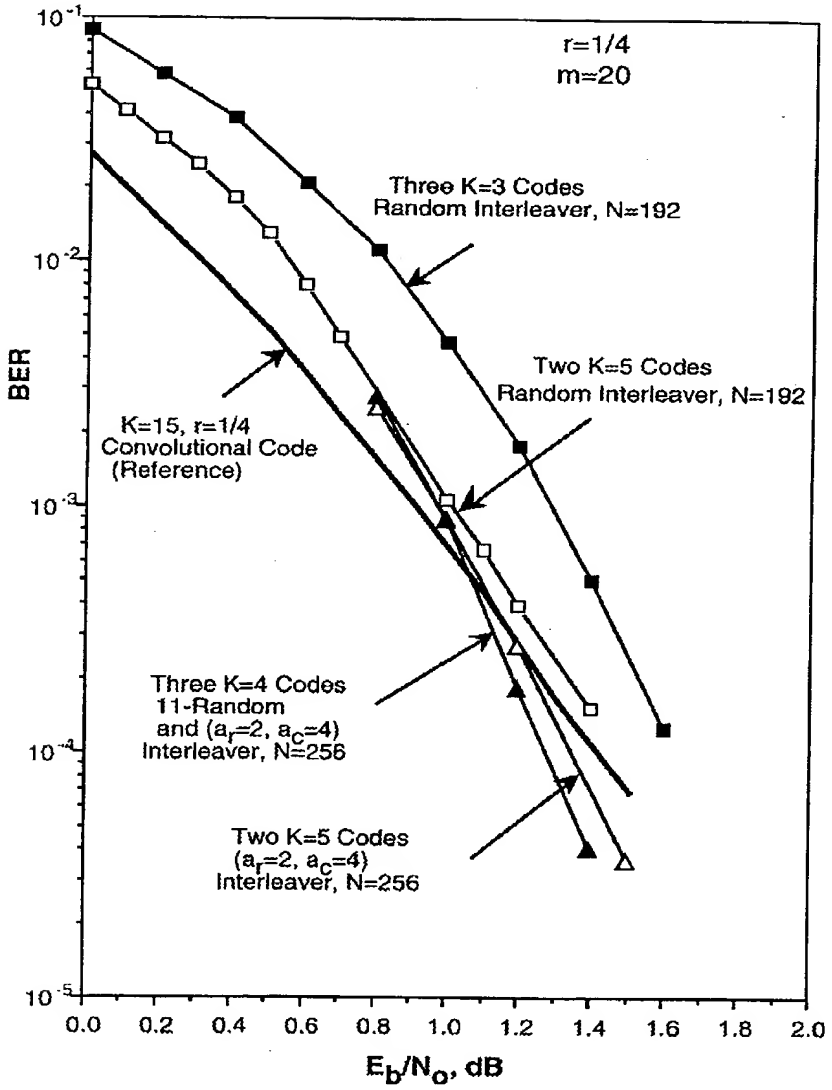

r=1/4
m=20
Three K=3 Codes
Random Interleaver, N=192
Two K=5 Codes
Random Interleaver, N=192
K=15, r=1/4
Convolutional Code
(Reference)
Three K=4 Codes
11-Random
and (a_r=2, a_c=4)
Interleaver, N=256
Two K=5 Codes
(a_r=2, a_c=4)
Interleaver, N=256
BER
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
E_b/N_o, dB

Figure 41

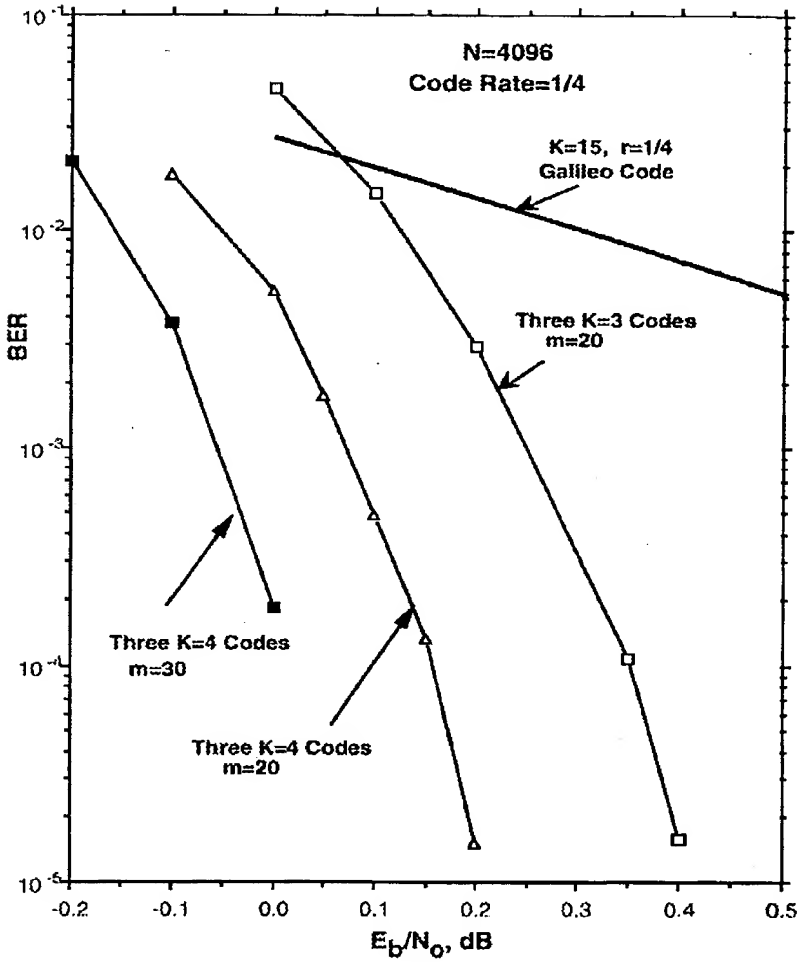

N=4096
Code Rate=1/4
K=15, r=1/4
Galileo Code
Three K=3 Codes
m=20
Three K=4 Codes
m=30
Three K=4 Codes
m=20
BER
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
-0.2
-0.1
0.0
0.1
0.2
0.3
0.4
0.5
E_b/N_o, dB

Figure 42

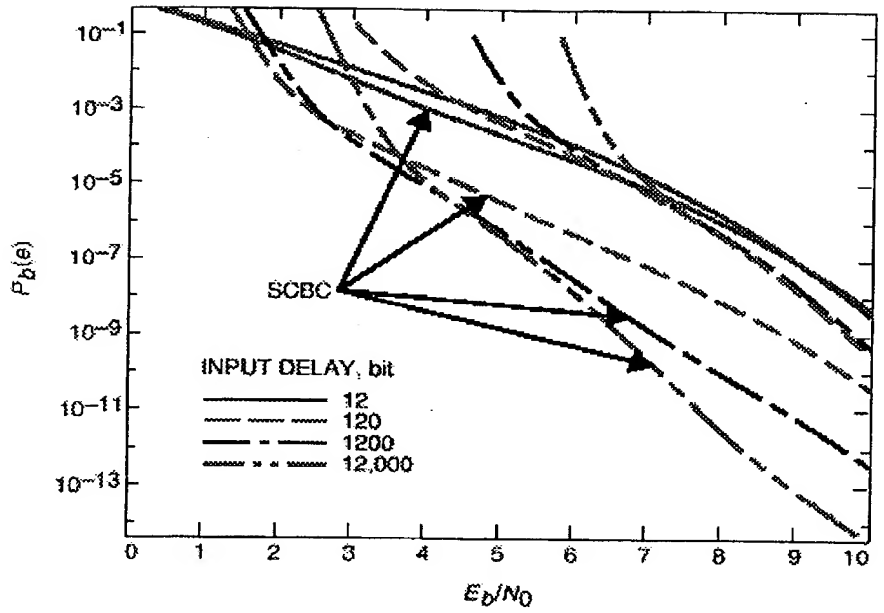


Figure 43

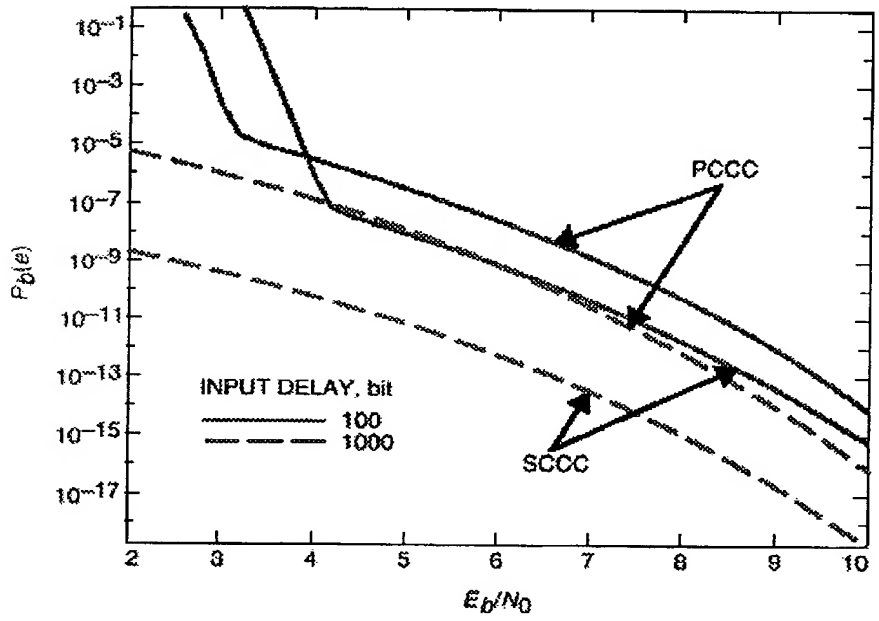


Figure 44

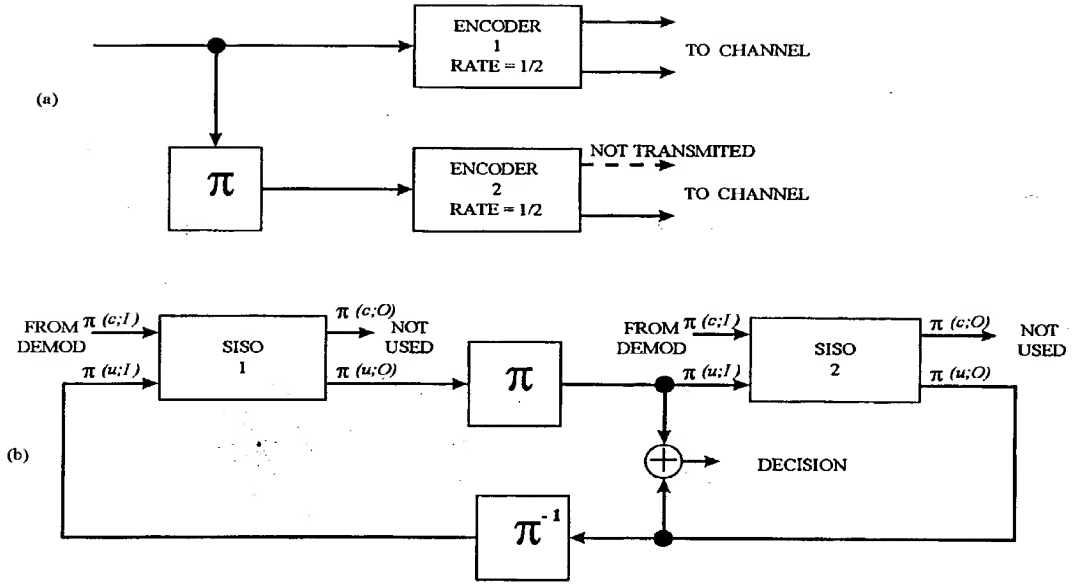


Figure 45

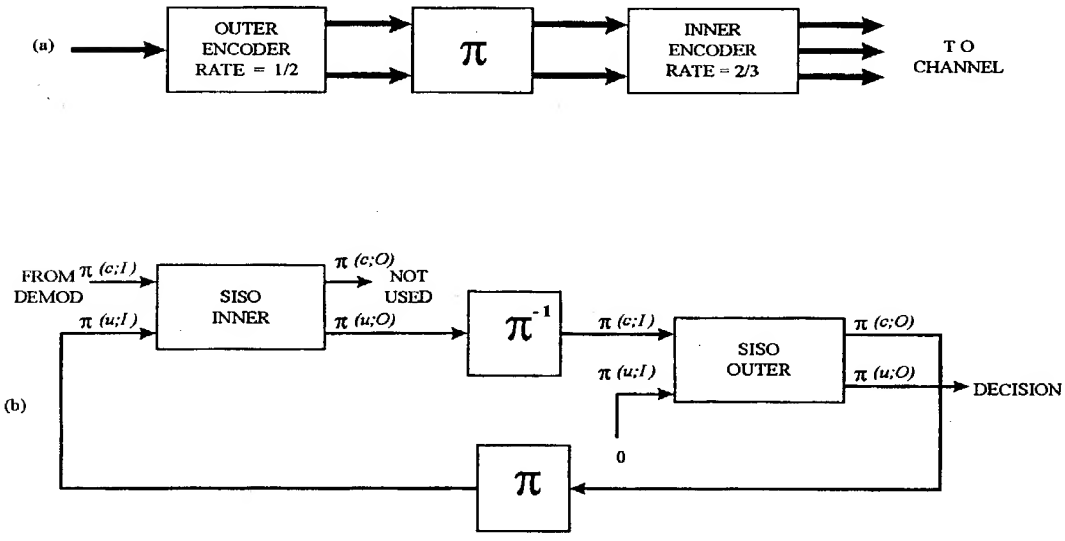


Figure 46

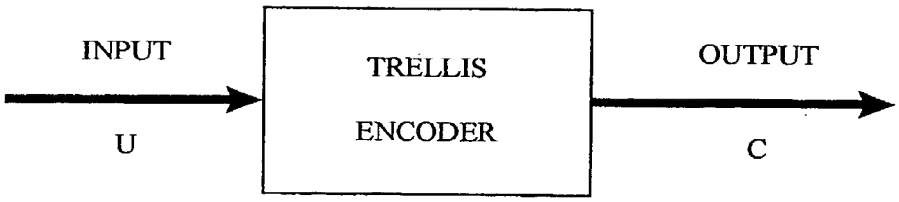


Figure 47

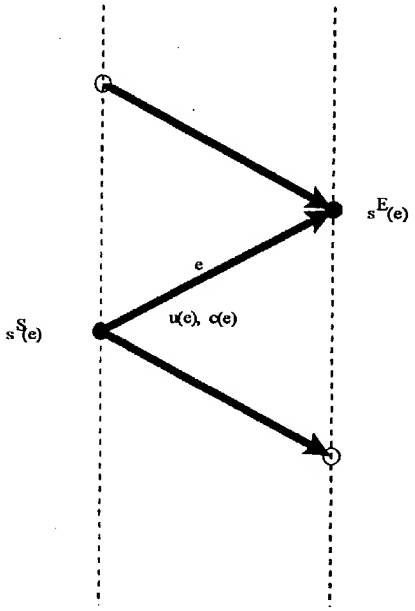


Figure 48

Figure 49

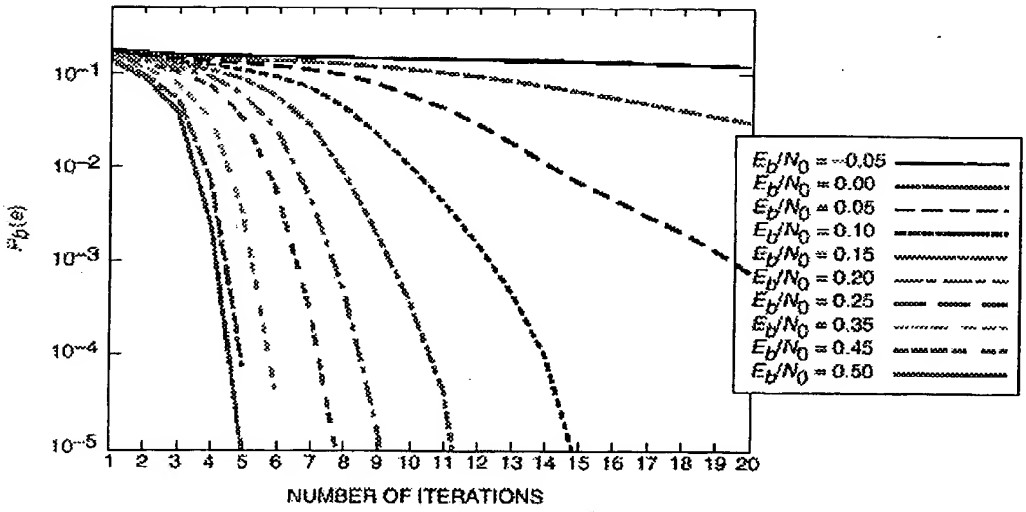


Figure 50

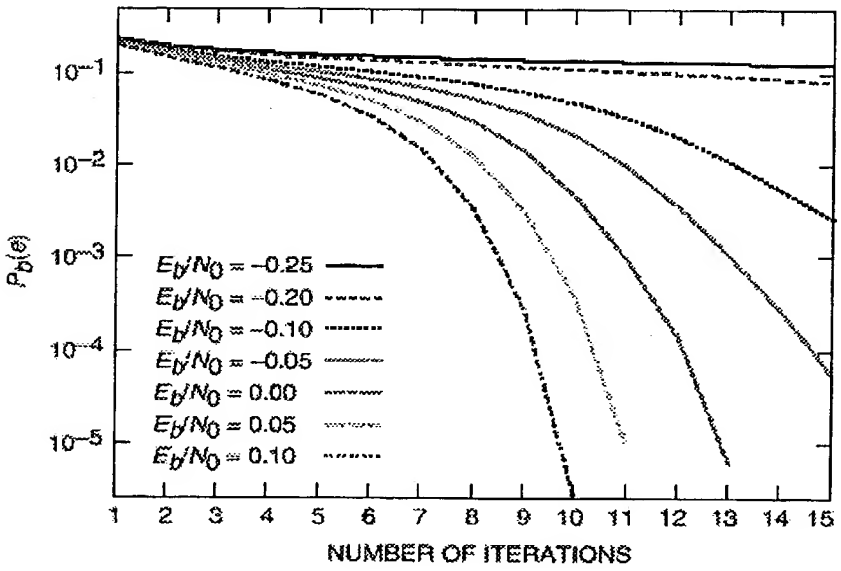


Figure 51

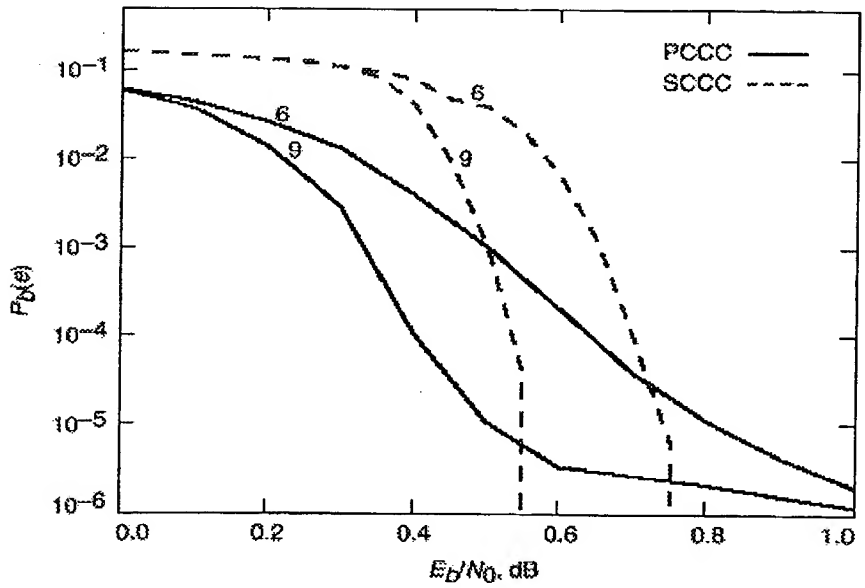


Figure 52

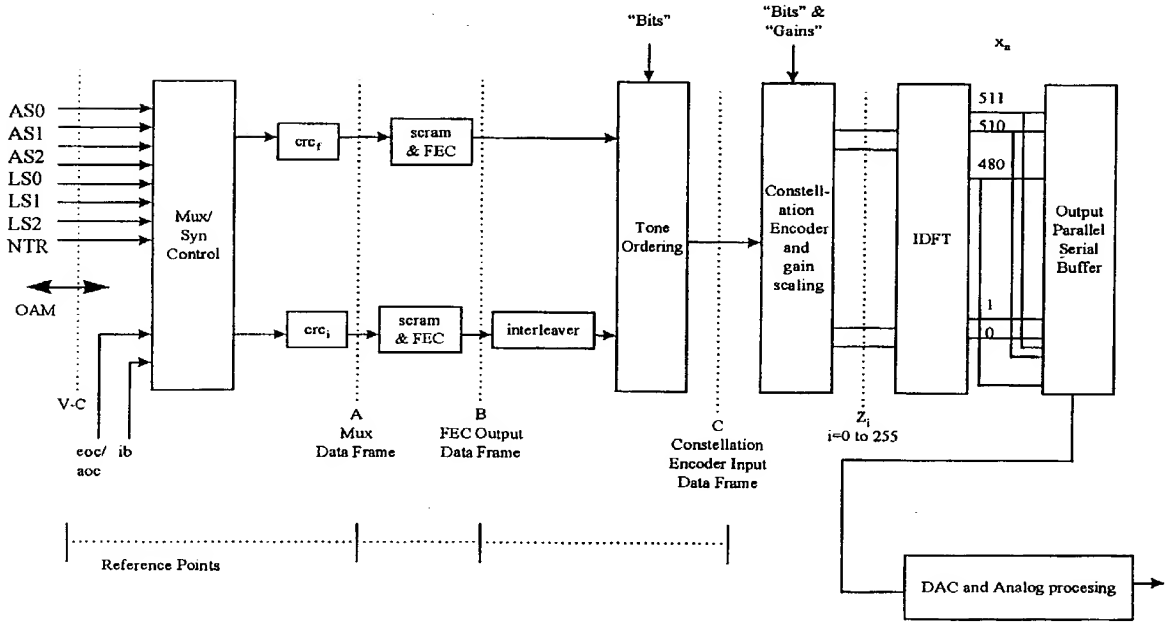


Figure 53

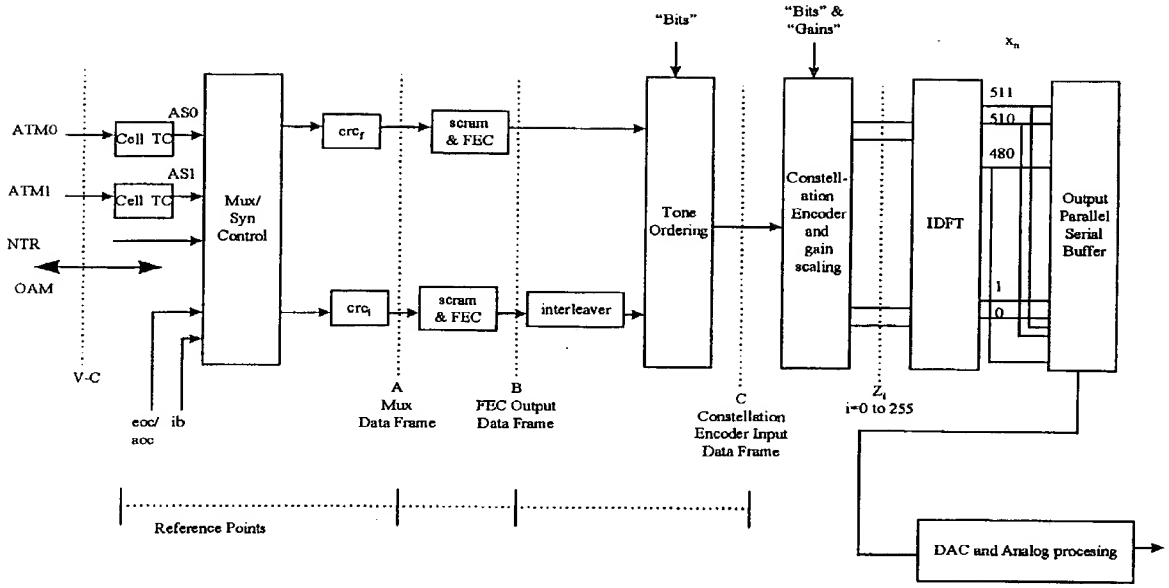


Figure 54

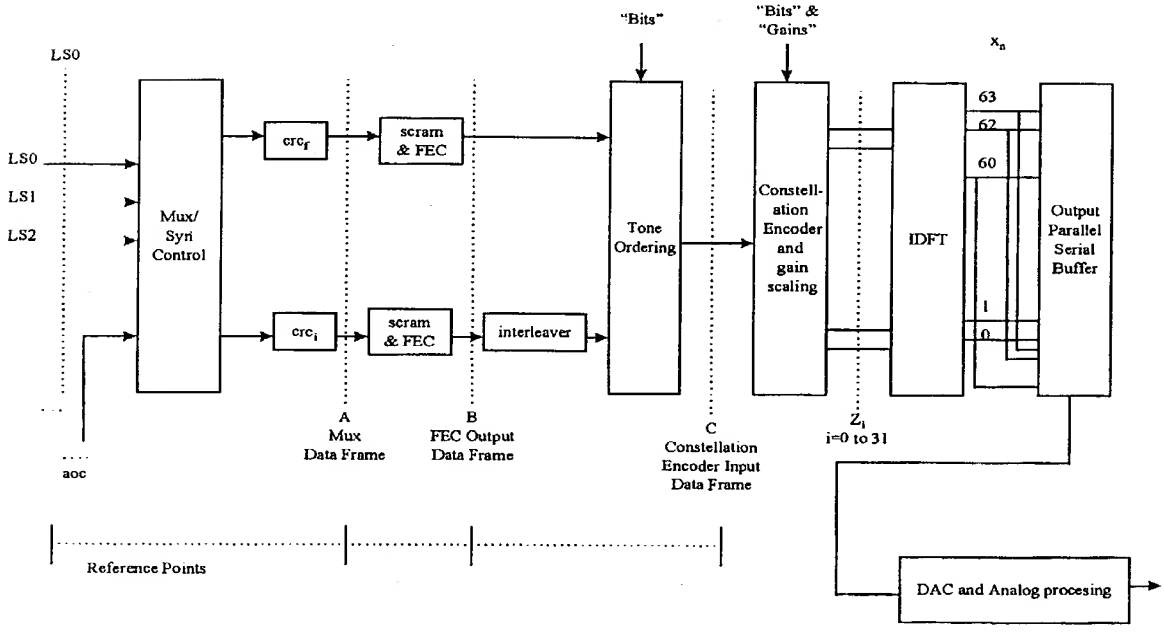


Figure 55

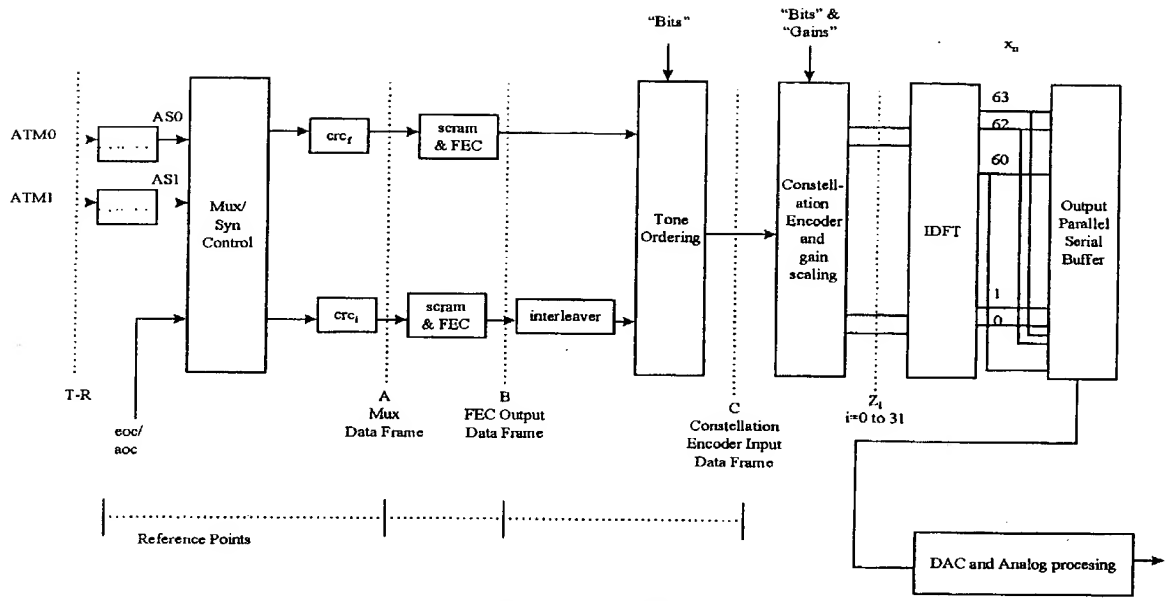


Figure 56

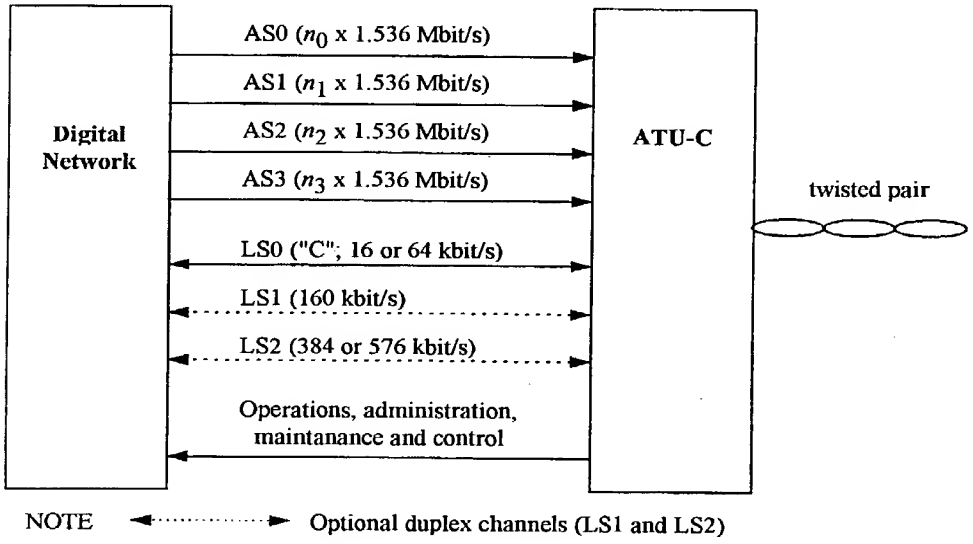


Figure 57

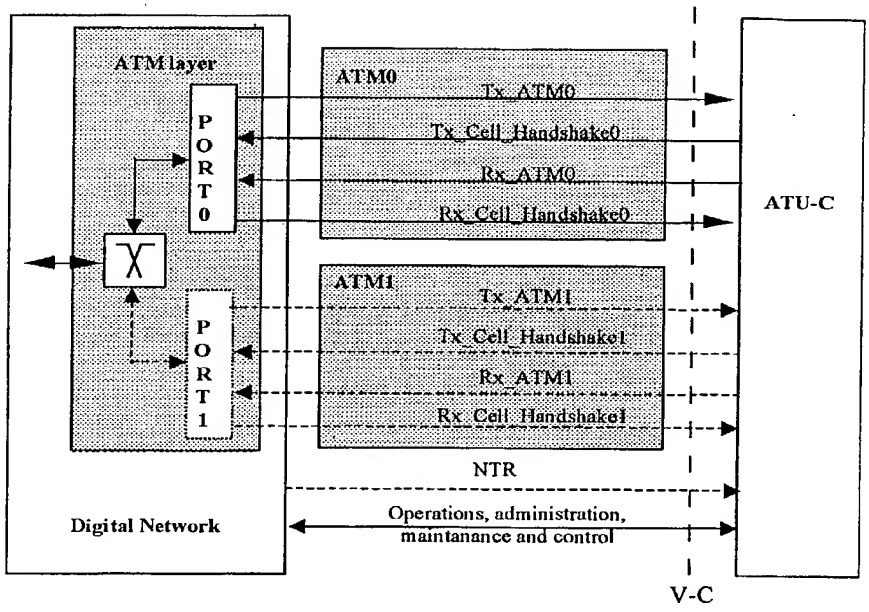


Figure 58

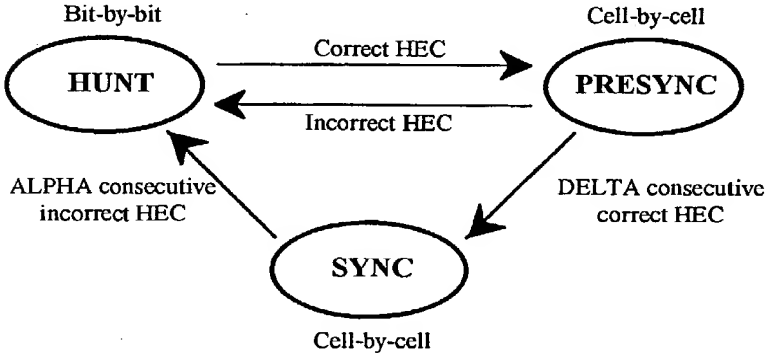
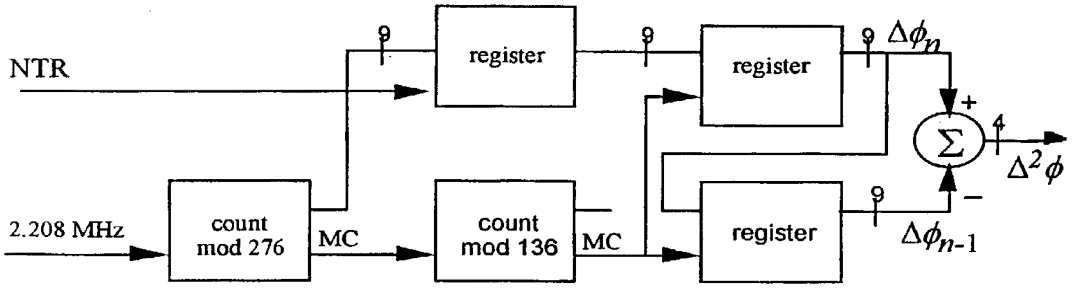


Figure 59



(MC is maxcount indication before foldover to 0)

Figure 60

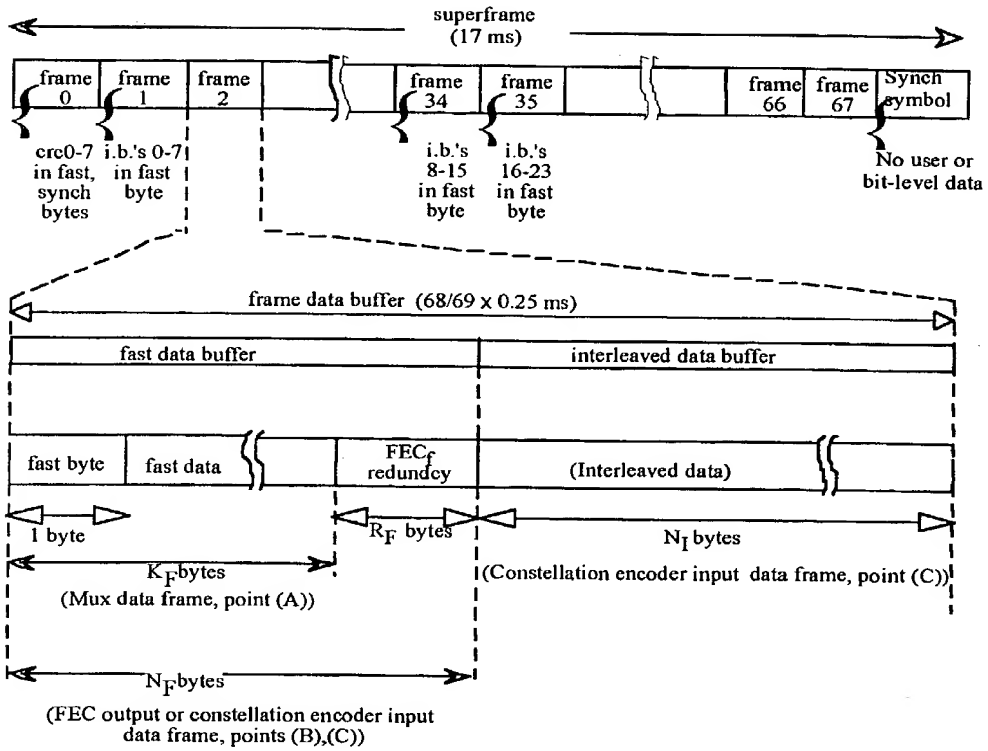
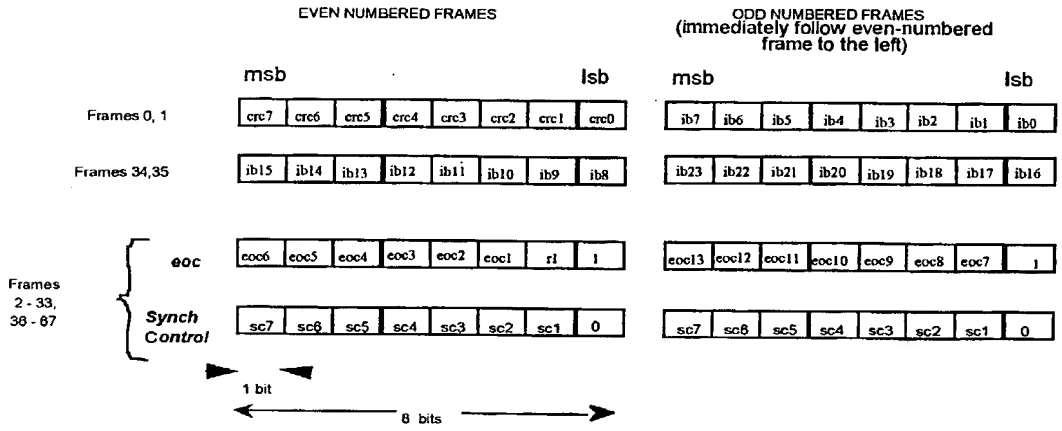


Figure 61



In all frames bit 7 = MSB and bit 0 = LSB

Figure 62

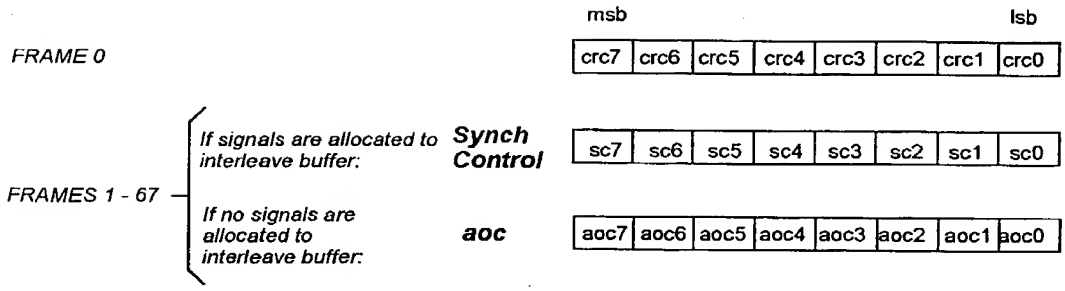


Figure 63

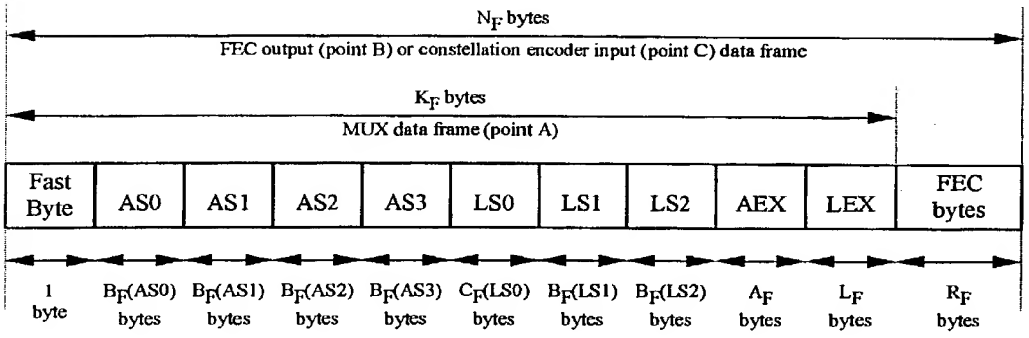


Figure 64

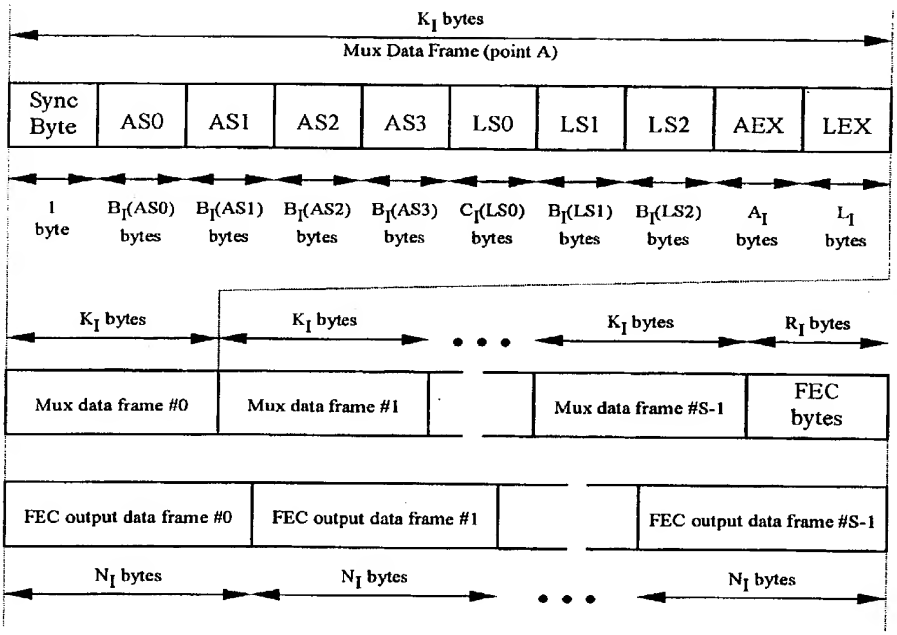


Figure 65

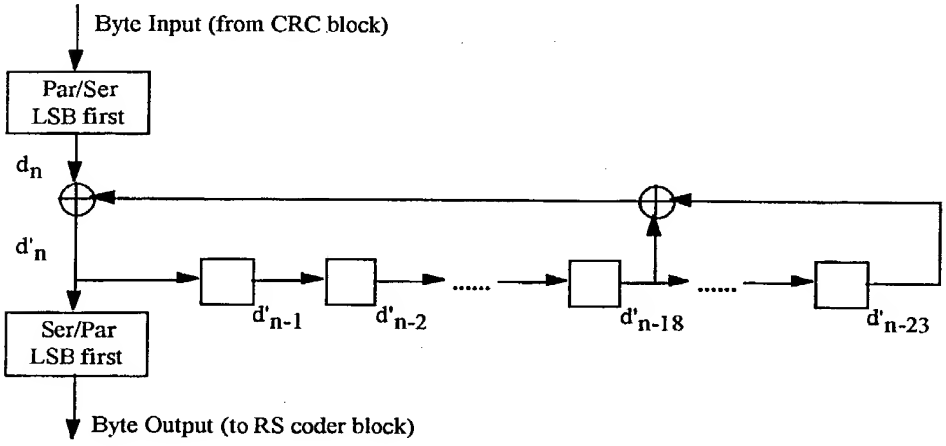


Figure 66

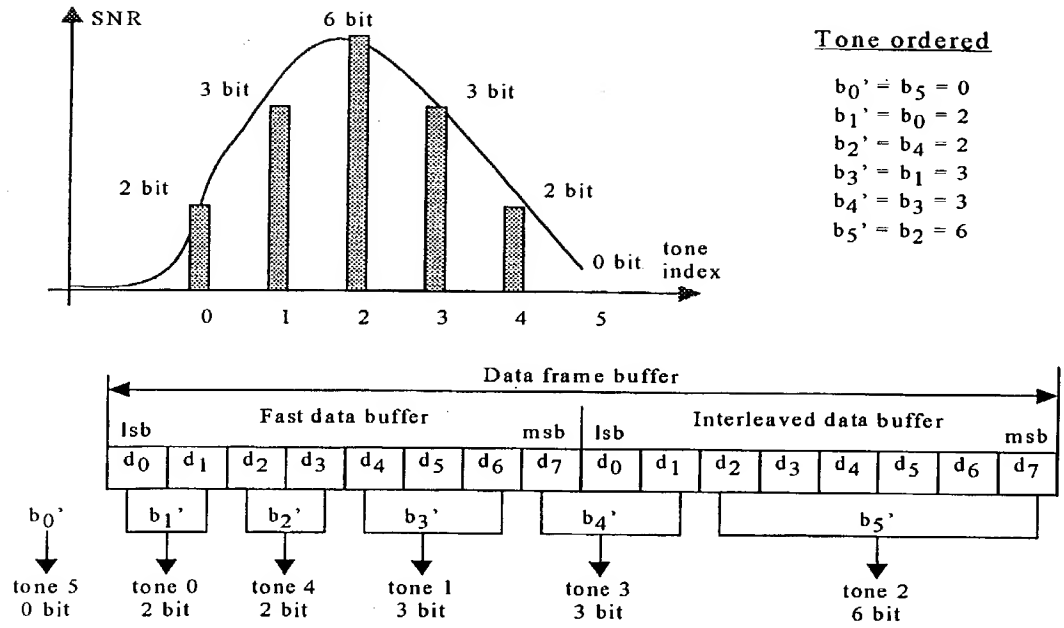


Figure 67

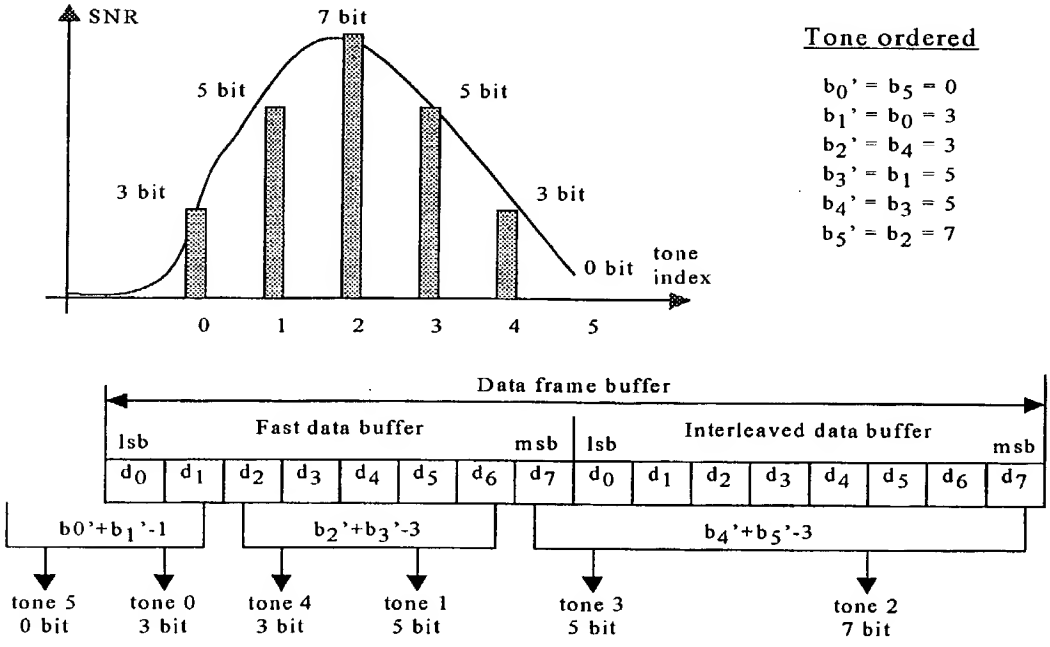


Figure 68

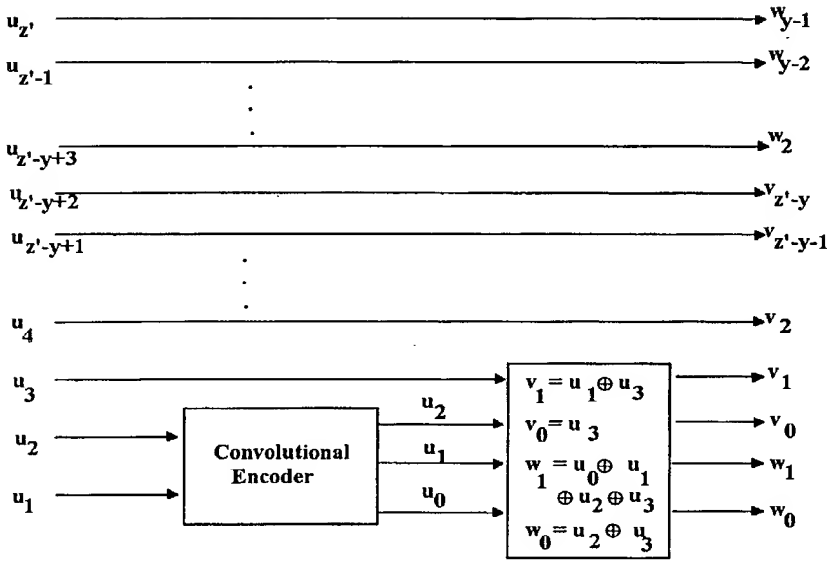


Figure 69

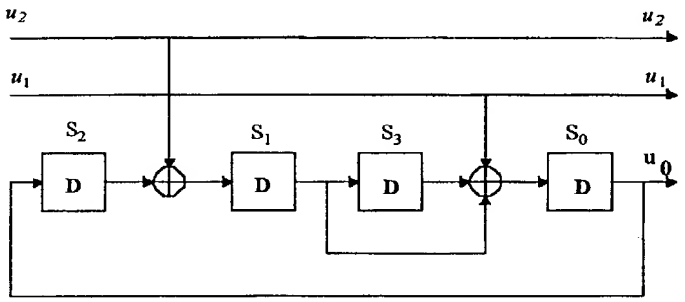


Figure 70

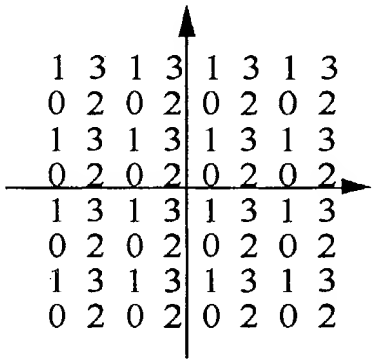


Figure 71

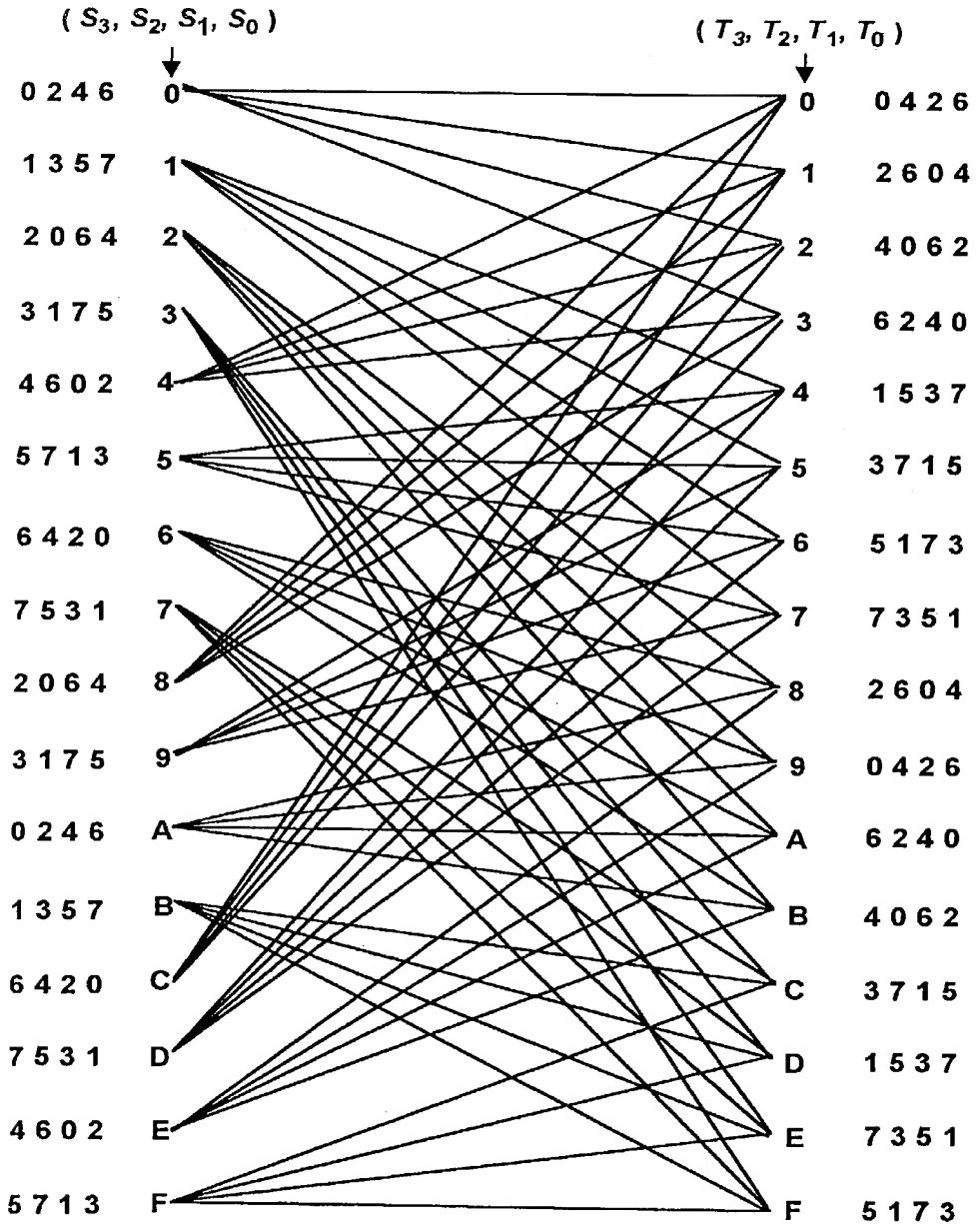


Figure 72

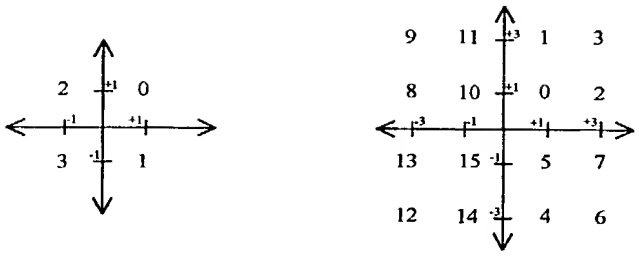


Figure 73

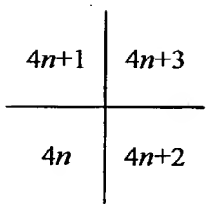


Figure 74

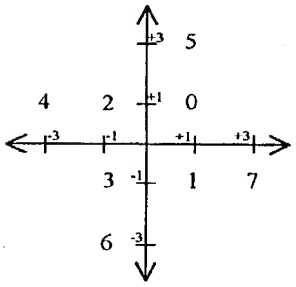


Figure 75

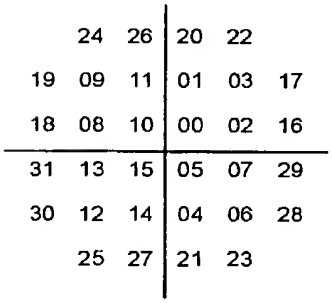


Figure 76

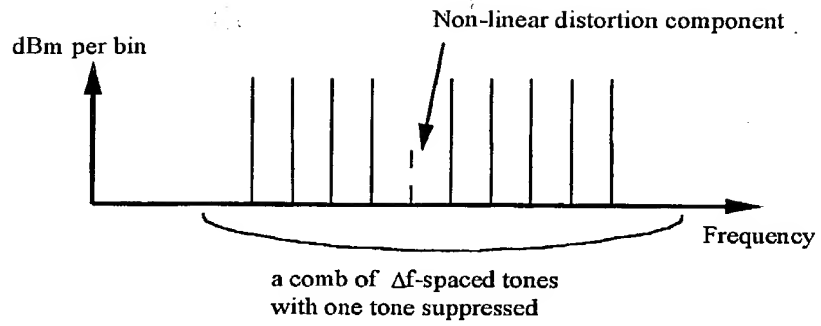


Figure 77

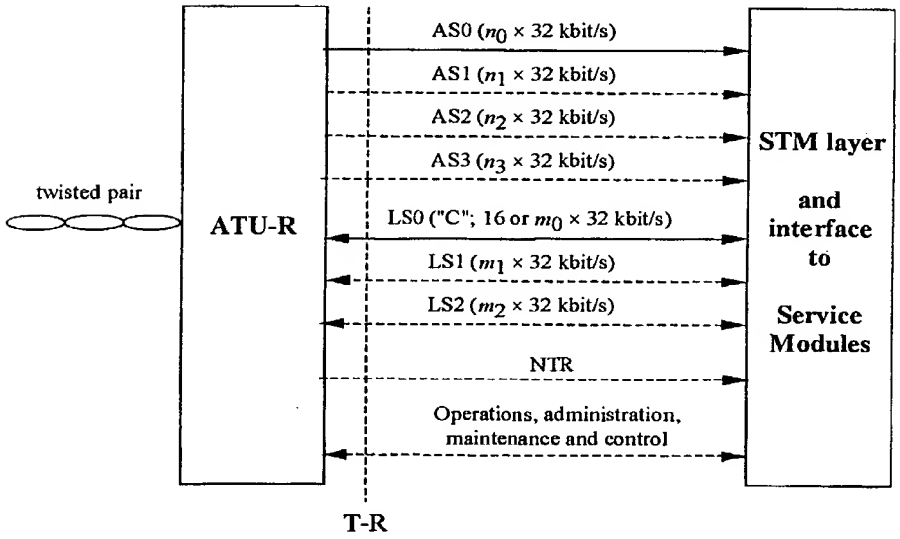


Figure 78

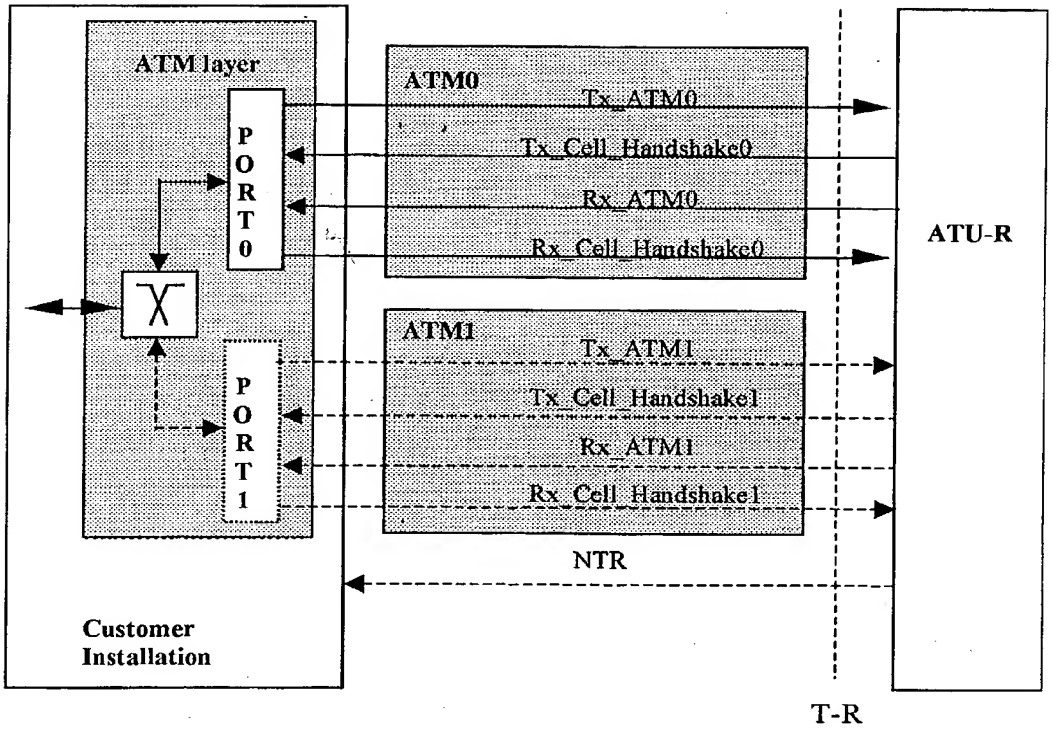


Figure 79

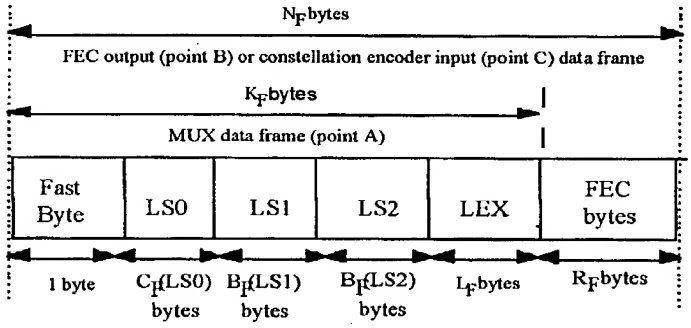


Figure 80

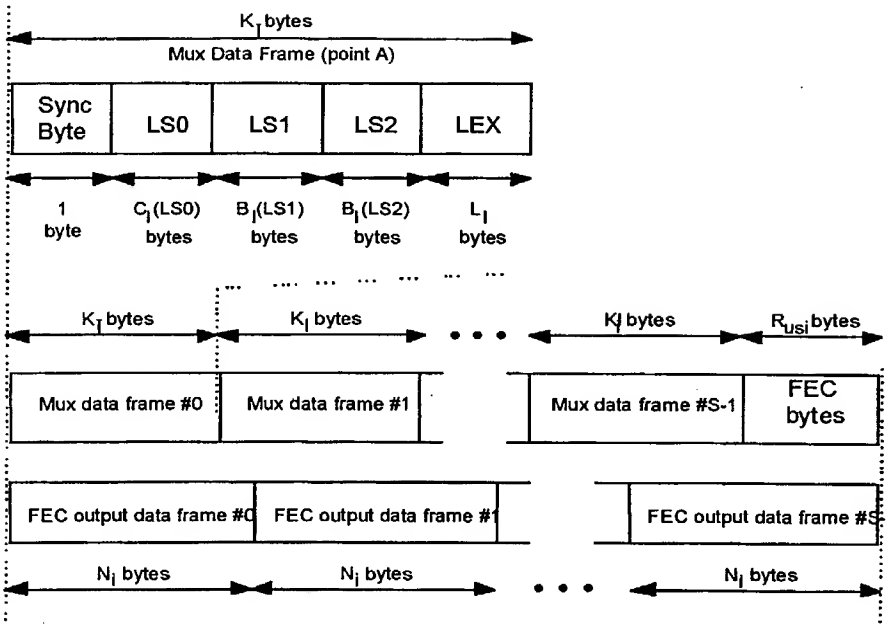


Figure 81

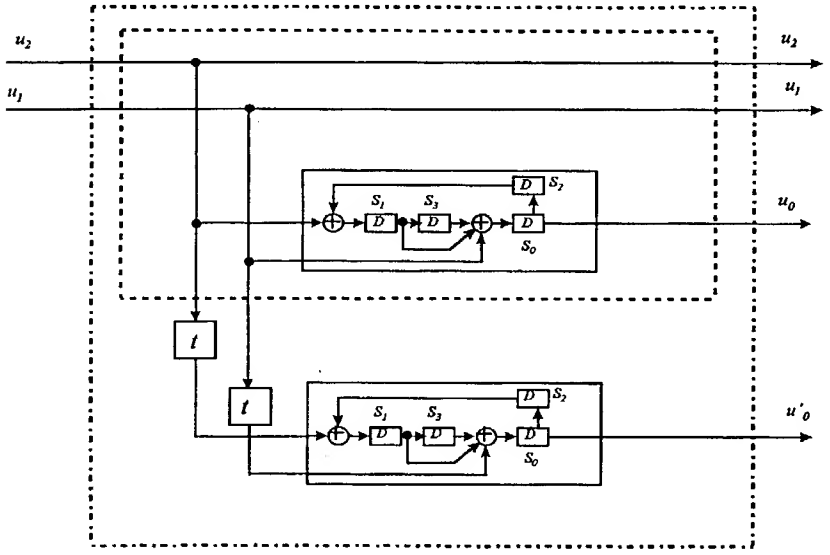


Figure 82

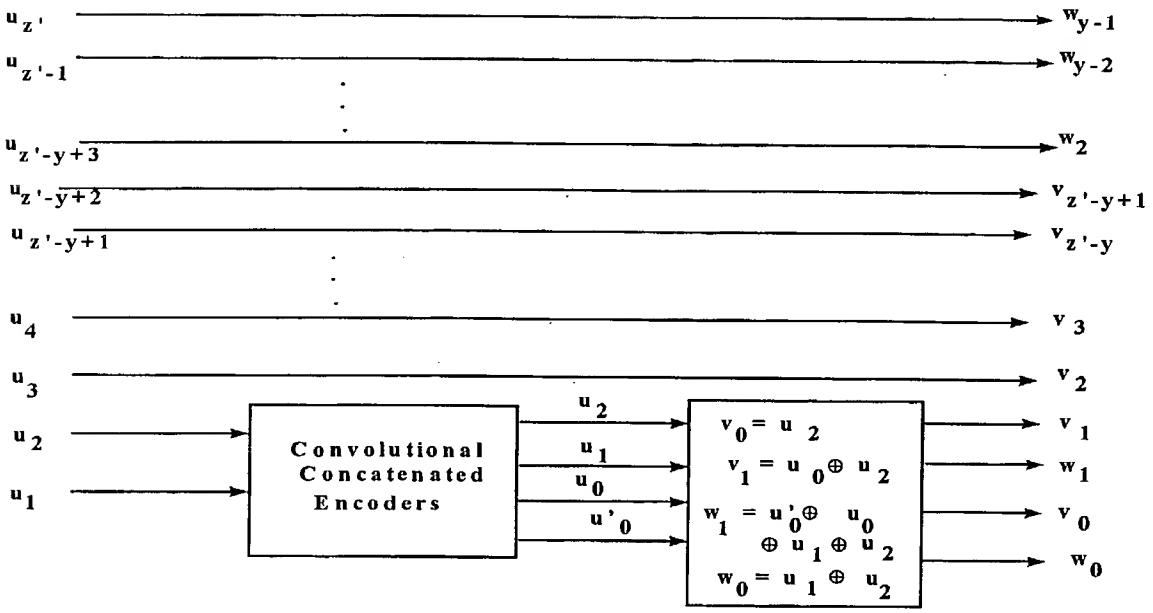


Figure 83

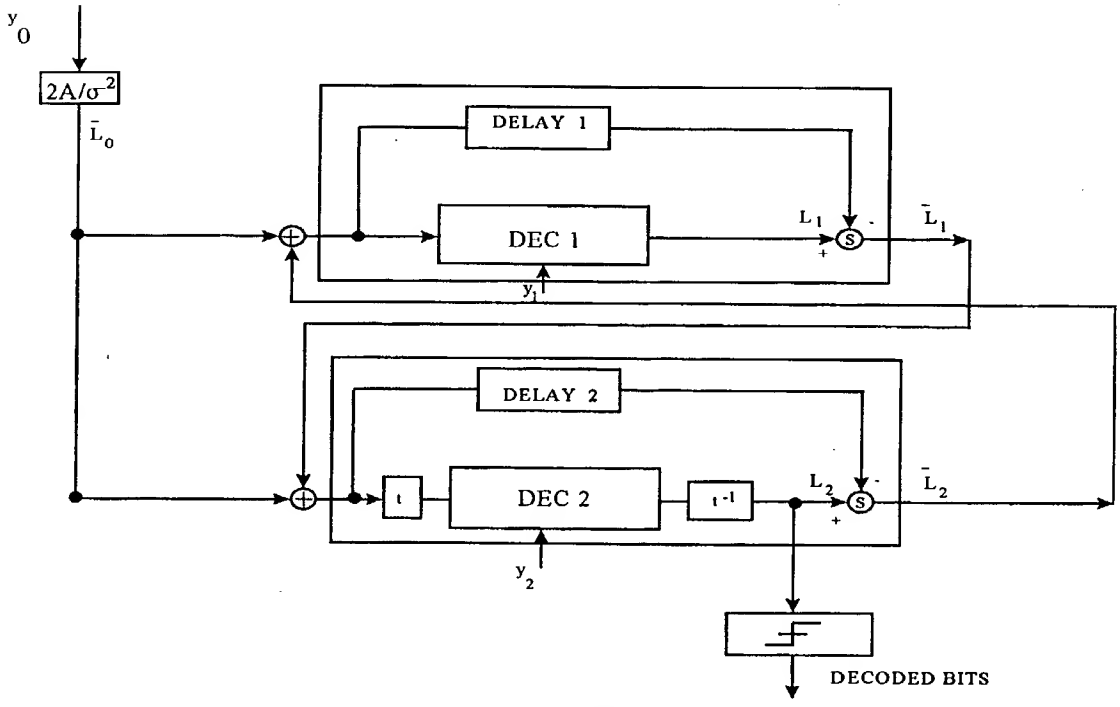


Figure 84

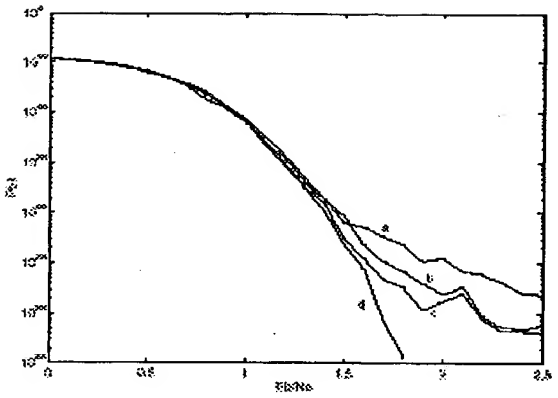


Figure 85

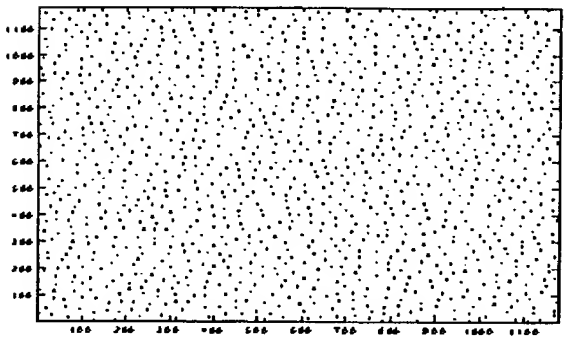


Figure 86

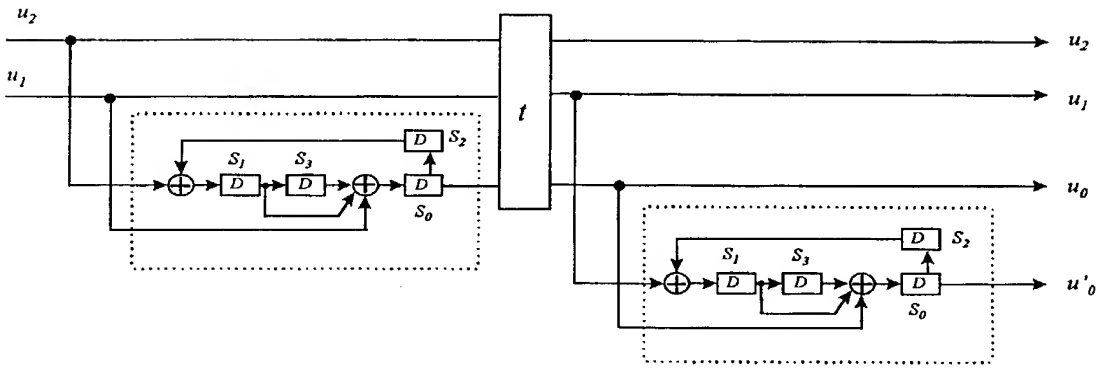


Figure 87

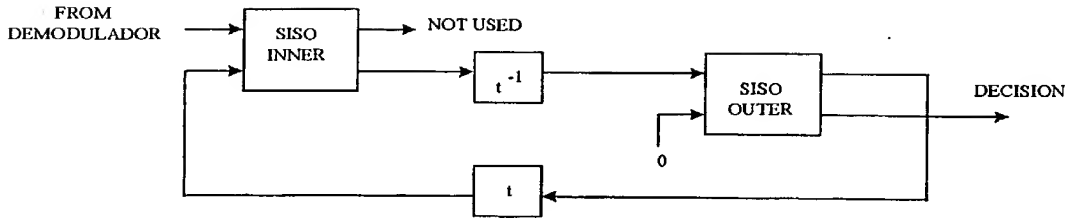


Figure 88

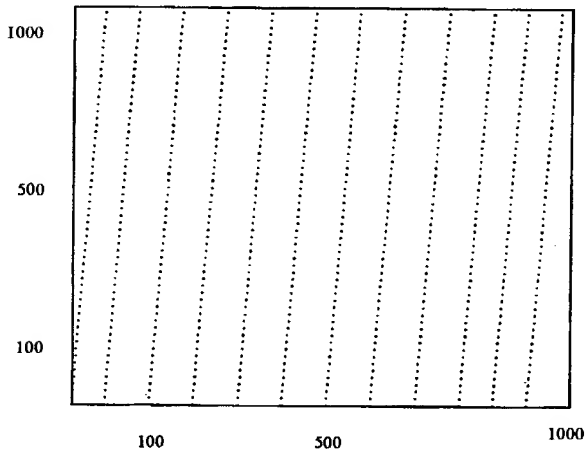


Figure 89

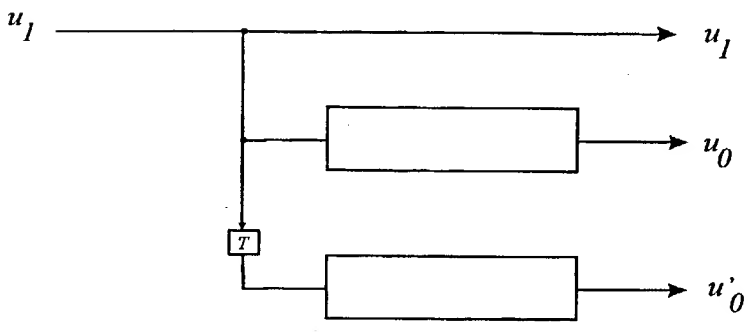


Figure 90

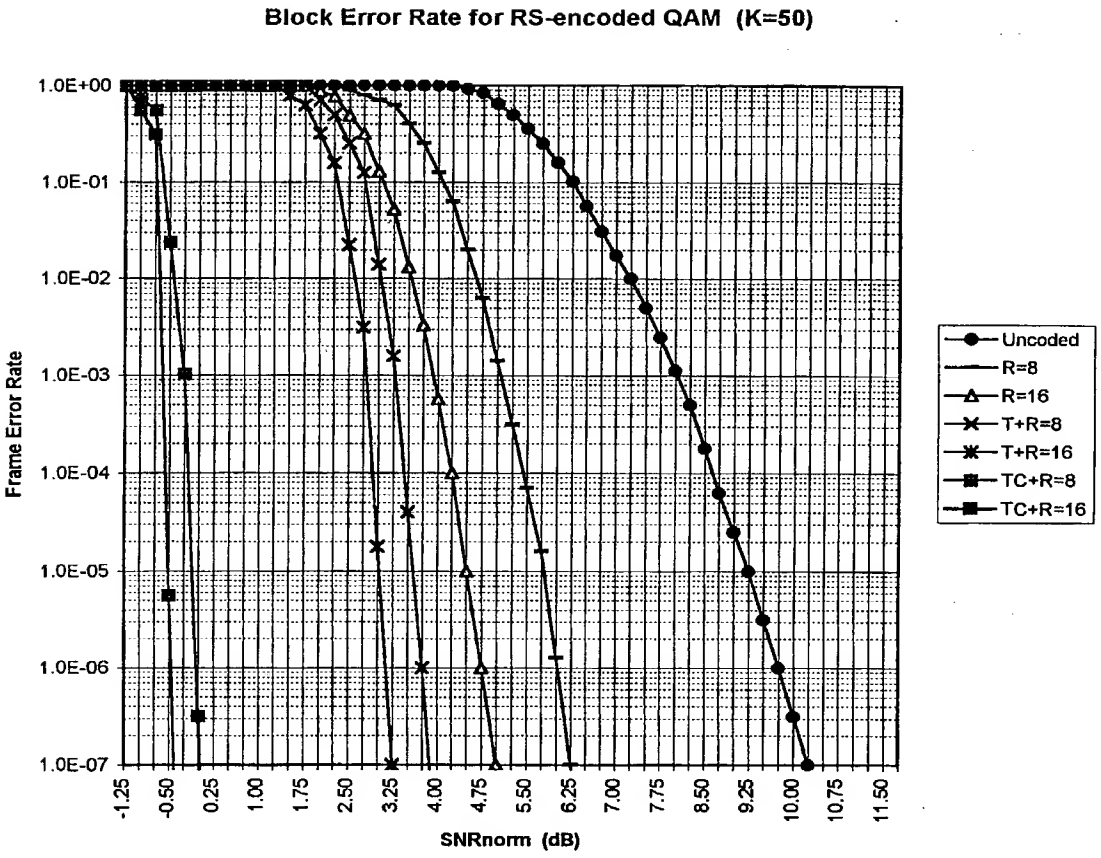


Figure 91

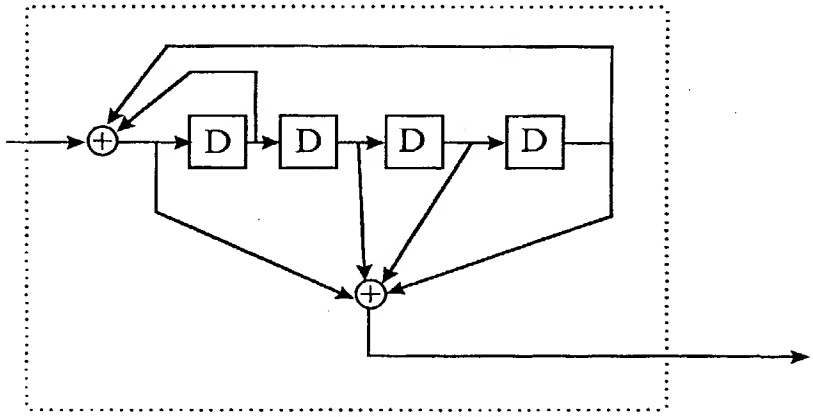


Figure 92

Docket No. 042159/0117

DECLARATION AND POWER OF ATTORNEY

As below named inventor, I hereby declare: THAT my residence, post office address and citizenship is as stated below next to my name; THAT we verily believe I am the original, inventor of the invention entitled

METHOD AND APPARATUS FOR DESIGN FORWARD ERROR CORRECTION TECHNIQUES IN DATA TRANSMISSION OVER COMMUNICATION SYSTEM

the specification of which:

[ ] is attached hereto.
[X] was filed on 30 January 2001 as Application Serial No. 09/744,790

[ ] is amended in the attached Amendment.

THAT we/I do not know and do not believe that this invention was ever known or used in the United States of America before our invention or discovery thereof, or patented or described in any printed publication in any country before our invention or discovery thereof, or more than one year prior to this application;

THAT the invention was not in public use or on sale in the United States of America for more than one year prior to this application;

THAT this invention has not been patented or made the subject of an inventor's certificate issued before the date of this application in any country foreign to the United States of America on an application filed by me or my legal representatives or assigns more than twelve months before this application;

THAT we/I have reviewed and understand the contents of the above identified specification, including the claim(s), as amended by any amendment referred to above;

THAT we/I acknowledge the duty to disclose information of which we are aware which is material to the examination of this application in accordance with 37 CFR §1.56; and

I HEREBY CLAIM foreign priority benefits under Title 35, United States Code §119(a)-(d) or §365(b) of any foreign application(s) for patent or inventor's certificate, or §365(a) of any PCT international application which designated at least one country other than the United States of America, listed below and have also identified below any foreign application for patent or inventor's certificate or of any PCT international application having a filing date before that of the application on which priority is claimed.

Prior Foreign Application Number	Country	Foreign Filing Date	Priority Claimed?	Certified Copy Attached?
PCT/US99/17369	PCT	July 30, 1999	Yes	

I HEREBY CLAIM the benefit under Title 35, United States Code §119(e) of any United States provisional application(s) listed below.

Serial No. 09/744,790

U.S. Provisional Application Number	Filing Date
60/094,629	July 30, 1998
60/098,394	August 30, 1998
60/133,390	May 10, 1999

I HEREBY CLAIM the benefit under Title 35, United States Code, §120 of any United States application(s), or § 365(c) of any PCT international application designating the United States of America, listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT International application in the manner provided by the first paragraph of Title 35, United States Code, § 112, I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56 which became available between the filing date of the prior application and the national or PCT international filing date of this application.

U.S. Parent Application Number	PCT Parent Application Number	Parent Filing Date	Parent Patent Number

And I hereby appoint, as my attorneys to prosecute this application and to transact all business in the Patent and Trademark Office connected therewith and with the resulting patent; individually and collectively:

Foley & Lardner
2029 Century Park East, Suite 3500
Los Angeles, California 90067-3021

telephone number (310) 277-2223 (to whom all communications regarding the subject application are to be directed); and each attorney thereof named below with Registration Numbers, and of the same address:

David A. Blumenthal Reg. No. 26,257
Ronald Coslick Reg. No. 36,489

We declare further that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Serial No. 09/744,780

1-00

Inventor's Signature:
Date: JULY 29, 2003
Inventor's Name (Typed): Juan Alberto Torres
First Middle Initial Family Name
Citizenship: United States
Residence (City): 34 South Lane Orchard Park NY 14127 NY
(State/Foreign Country): United States
Post Office Address:

2-00

Inventor's Signature:
Date: July 29, 2003
Inventor's Name (Typed): Victor Demjanenko
First Middle Initial Family Name
Citizenship: United States
Residence (City): 4420 Beach Ridge Road Pendleton NY 14094 NY
(State/Foreign Country): United States
Post Office Address:

3-00

Inventor's Signature:
Date: JULY 29, 2002
Inventor's Name (Typed): Frederic Hirzel
First Middle Initial Family Name
Citizenship: United States
Residence (City): 1725 Edinborough Rochester Hills MI 48306 MI
(State/Foreign Country): United States
Post Office Address:

Computer Program Listing Appendix
(containing 37 pages of source code)

5-04-99 7:06p y:test

```
enable_scrambler          =      1;
enable_reed_solomon       =      0;
enable_interleaver        =      1;
enable_trellis_decoder    =      0;
enable_turbo_decoder      =      0;

print_internal_points     =      0;   #  1;
print_reed_solomon_data   =      0;   #  1;
show_energy_measurements  =      0;   #  1;

BLOCK_LENGTH              =   1000L;
ENERGY_LENGTH             =  10000L;
REPORT_LENGTH             =    100L;

DB_START                  =    0.0;
DB_STOP                   =   16.0;
DB_STEP                   =    1.0;

count_limit0              =     0L;
error_limit0              =  1000L;

count_limit1              =  5000000L;
error_limit1              =   100L;

count_limit2              =  1000000L;
error_limit2              =    10L;

count_limit3              =  5000000L;
error_limit3              =     5L;

count_limit4              =  7500000L;
error_limit4              =     3L;

count_limit5              =  10000000L;
error_limit5              =     0L;

TIMES                     =      12;

REED_SOLOMON_FRAME_SIZE   =      51;
REED_SOLOMON_CHECK_SYMBOLS =      0;

INTERLEAVER_FRAME_DEPTH   =      8;

TRELLIS_SYMBOL_SIZE       =      3;
TRELLIS_LOOKBACK_LENGTH   =     20;
TRELLIS_BLOCK_DELAY       =      1;

TURBO_FRAME_SIZE          =    128;
TURBO_ITERATION_COUNT     =     10;

##########################################################################################
```

5/04/99 3:38p y:MAKEFILE

```
all : test.exe x.exe

test.exe : test.cpp rs.cpp trellis.cpp scrmblr.cpp random.cpp parse.cpp turbo.cpp te
st.h
    bcc32 -w-csu test.cpp rs.cpp trellis.cpp scrmblr.cpp random.cpp parse.cpp turbo.
cpp
    rm *.bak *.obj

x.exe : x.cpp turbo.cpp
    bcc32 -w-csu x.cpp turbo.cpp
    rm *.bak *.obj
```

```
#define WE_ARE_DEFINING_VARIABLES

#include "test.h"

#define init_reedsolo       init_rs
#define encode_reedsolo     encode_rs
#define decode_reedsolo     eras_dec_rs

dtype *rs_data;

unsigned *tx_data;
unsigned *rx_data;
unsigned *rx_perf;

unsigned *tx_payload_data;
unsigned *rx_payload_data;
unsigned *rx_payload_perf;

unsigned *tx_frame_data;
unsigned *rx_frame_data;
unsigned *rx_frame_perf;

unsigned *tx_interleave_data [DD];
unsigned *rx_interleave_data [DD];
unsigned *rx_interleave_perf [DD];

unsigned *tx_trellis_data;
unsigned *rx_trellis_data;
unsigned *rx_trellis_perf;

unsigned *tx_trellis_dely;
unsigned *rx_trellis_dely;

//
//  define setup pointers
//

int tx_frame_ptr;
int rx_frame_ptr;

int tx_trellis_dely_ptr;
int rx_trellis_dely_ptr;

//
//  define error reporting variables
//

long total_error_blocks;
long total_count_blocks;
long total_error_symbols;
long total_count_symbols;

#define PRINT_POINTS    if (print_internal_points)
#define PRINT_REED_SOLO if (print_reed_solomon_data)
#define SHOW_ENERGY     if (show_energy_measurements)

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

#define INIT(a,s,v) INIT1 (a,s,v)

#define INIT1(a,s,v) (int i; for(i=0;i<s;i++) a[i]=v;)
#define INIT2(a,s1,s2,v) (int i,j; for(i=0;i<s1;i++) for(j=0;j<s2;j++) a[i][j]=v;)

double
dB (double x)
{
    if (x < 1e-10) x = 1e-10;
    return 10.*log10(x);
}

#define ALLOC(X,Y,Z)    if ((Y = (X *) malloc (sizeof (X) * Z)) == NULL) (fprintf (s
tderr, "error: could not allocate memory\n"); exit (1);)

void
alloc_data (void)
{
    int i;

    ALLOC (dtype, rs_data, NN);

    ALLOC (unsigned, tx_data, N);
    ALLOC (unsigned, rx_data, N);
    ALLOC (unsigned, rx_perf, N);

    ALLOC (unsigned, tx_payload_data, N);
    ALLOC (unsigned, rx_payload_data, N);
    ALLOC (unsigned, rx_payload_perf, N);

    ALLOC (unsigned, tx_frame_data, N);
    ALLOC (unsigned, rx_frame_data, N);
    ALLOC (unsigned, rx_frame_perf, N);

    for (i = 0; i < DD; i++) {
    ALLOC (unsigned, tx_interleave_data [i], N);
    ALLOC (unsigned, rx_interleave_data [i], N);
    ALLOC (unsigned, rx_interleave_perf [i], N);
    }

    ALLOC (unsigned, tx_trellis_data, T);
    ALLOC (unsigned, rx_trellis_data, T);
    ALLOC (unsigned, rx_trellis_perf, T);

    ALLOC (unsigned, tx_trellis_dely, B*T);
    ALLOC (unsigned, rx_trellis_dely, B*T);
}

void
init_data (void)
{
    INIT (rs_data, NN, 0);

    INIT (tx_data, K, 0);
    INIT (rx_data, K, 0);
    INIT (rx_perf, K, 0);

    INIT (tx_payload_data, K, 0);
    INIT (rx_payload_data, K, 0);
    INIT (rx_payload_perf, K, 0);

    INIT (tx_frame_data, N, 0);
```


WO 00/07323 PCT/US99/17369

```
    INIT (rx_frame_data, N, 0);
    INIT (rx_frame_perf, N, 0);

    INIT2 (tx_interleave_data, 0, N, 0);
    INIT2 (rx_interleave_data, 0, N, 0);
    INIT2 (rx_interleave_perf, 0, N, 0);

    INIT (tx_trellis_data, 1, 0);
    INIT (rx_trellis_data, 1, 0);
    INIT (rx_trellis_perf, 1, 0);

    INIT (tx_trellis_dely, B*T, 0);
    INIT (rx_trellis_dely, B*T, 0);

    tx_frame_ptr = 0;
    rx_frame_ptr = 0;

    tx_trellis_dely_ptr = 0;
    rx_trellis_dely_ptr = 0;

    total_error_blocks  = 0L;
    total_count_blocks  = 0L;
    total_error_symbols = 0L;
    total_count_symbols = 0L;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

static double dB_signal = 0.0;
static double dB_noise  = 0.0;
static double dB_count  = 0.0;
static long   dB_energy = 0L;

void
main (int argc, char *argv [])
{
    unsigned seed;
    long block;
    double SN;

    spcout = fopen ("output.out", "w");
    if (spcout == NULL) { fprintf (stderr, "error: could not open file %s\n", "outpu
t.out"); exit (1); }

    argc = argc;
    argv = argv;

    seed = (unsigned) time (NULL);
    srand (seed);

    initialize_options ();
    alloc_data ();

    alloc_turbo_memory (TURBO_FRAME_SIZE, TURBO_ITERATION_COUNT);

for (SN = DB_START; SN <= DB_STOP+1e-10; SN += DB_STEP) {
    SIGMA = pow (10., -SN/20);

    init_data ();
    init_random (SIGMA, TIMES);
    init_scrambler ();
    init_reedsolo ();
    init_trellis ();
    init_turbo (SIGMA);

    dB_signal = 0.0;
    dB_noise  = 0.0;
    dB_count  = 0.0;
    dB_energy = 0L;

    for (block = 0L; block < BLOCK_LENGTH; block++) {

        //
        //  create data of K symbols of M bits/symbol
        //

        {
            int i;

            for (i = 0; i < K; i++)
                tx_data [i] = rand () & ((1<<M)-1);
        }

        //
        //  scramble payload of K symbols of M bits/symbol
        //

        {
            int i;
            static long symbol = 0L;

            for (i = 0; i < K; i++) {
                if (!enable_scrambler)  tx_payload_data [i] = tx_data [i];
                else                    tx_payload_data [i] = scrambler_encode (tx_d
ata [i]);

                PRINT_POINTS {
                    if (block < BLOCK_LENGTH-B-(0-1)) {
                        fprintf (spcout, "%8ld tx_data %04x\n", symbol, tx_data [i])
;
                        fprintf (spcout, "%8ld tx_payload_data %04x\n", symbol, tx_p
ayload_data [i]);
                        symbol++;
                    }
                }
            }
        }

        //
        //  reed-solomon encode frame of N symbols of M bits/symbol
        //

        {
            int i;

            for (i = 0; i < NN; i++)
                rs_data [i] = 0;

            for (i = 0; i < K; i++)
                rs_data [i] = tx_payload_data [i];
```

```
            if (!enable_reed_solomon)   ;
            else                        encode_reedsolo (&rs_data [0], &rs_data [KK]
);

            for (i = 0; i < K; i++)
                tx_frame_data [i] = rs_data [i];

            for (i = 0; i < R; i++)
                tx_frame_data [K+i] = rs_data [KK+i];

            PRINT_REED_SOLO (
                if (enable_reed_solomon) {
                    int error = decode_reedsolo (rs_data, NULL, 0);
                    fprintf (spcout, "%8ld : %8d ... %d %d %d %d\n", block, error, M
M, NN, KK, RR);
                    for (i = 0; i < NN; i++) fprintf (spcout, "%4d %8d\n", i, rs_dat
a [i]);
                }
            )
        }

        //
        //  convolutionally interleave D depth N symbols of M bits/symbol
        //
        //  NOTE: N must be ODD
        //

        {
            int i;
            static long symbol = 0L;
            int idx = 0;
            int frm = tx_frame_ptr;

            for (i = 0; i < N; i++) {
                tx_interleave_data [frm] [idx] = tx_frame_data [i];
                idx = idx + D;
                if (idx >= N) {
                    idx = idx - N;
                    if (++frm >= D) frm = 0;
                }

                PRINT_POINTS (
                    if (block < BLOCK_LENGTH-B-(D-1)) {
                        fprintf (spcout, "%8ld tx_frame_data %04x\n", symbol, tx_fra
me_data [i]);
                        fprintf (spcout, "%8ld tx_interleave_data %04x\n", symbol, t
x_interleave_data [tx_frame_ptr] [i]);
                    symbol++;
                    }
                )
            )
        }

        //
        //  create output of M*N/U symbols of U bits/symbol
        //
        //      U = 4i+3bits; i int
        //
        //  NOTE: N most be multiple of U
        //
```

```
        {
            int i;
            int idx = 0;
            int bit = 0;
            unsigned dat = 0x00000000L;

            for (i = 0; i < T; i++) {
                while (bit < U) {
                    dat = dat | (tx_interleave_data [tx_frame_ptr] [idx]<<bit);
                    bit = bit + M, idx++;
                }
                tx_trellis_data [i] = dat & ((1<<U)-1);
                dat = dat >> U;
                bit = bit - U;
            }
            if (++tx_frame_ptr >= D) tx_frame_ptr = 0;
        }

/*??????????????????????????????????????????????????????????????????????????????*/
    if (enable_turbo_decoder) {
/*??????????????????????????????????????????????????????????????????????????????*/

        {
            int i;
            int j;
            double no_A;
            double no_B;
            double no_C;

            for (i = 0; i < T; i++) {

                rx_trellis_perf [i] = 0;
                rx_trellis_data [i] = 0;

            for (j = 0; j < U; j++) {

                tx_bits = (tx_trellis_data [i] >> j) & 0x01;

                turbo_encode ();

                no_A = random_value ();
                no_B = random_value ();
                no_C = random_value ();

                in_A = tx_A + no_A;
                in_B = tx_B + no_B;
                in_C = tx_C + no_C;

            #define BLIND_FLAG  0

                turbo_decode (BLIND_FLAG);

                rx_trellis_perf [i] |= (dl_bits << j);
                rx_trellis_data [i] |= (rx_bits << j);

                SHOW_ENERGY (
                    dB_count  += 3.0;
                    dB_signal += tx_A*tx_A + tx_B*tx_B + tx_C*tx_C;
                    dB_noise  += no_A*no_A + no_B*no_B + no_C*no_C;

                    dB_energy++;
```

```
                if ((dB_energy%ENERGY_LENGTH) == 0) {
                //  fprintf (spcout, "%8ld - %10.2f %10.2f %10.2f\n"
                //      , dB_energy
                //      , dB (dB_signal/dB_count)
                //      , dB (dB_noise /dB_count)
                //      , dB (dB_noise /dB_signal)
                //  );
                    fprintf (spcout, "%8ld - %10.2f\n", dB_energy, dB (dB_noise
/dB_signal));
                    dB_signal = 0.0;
                    dB_noise  = 0.0;
                    dB_count  = 0.0;
                }
            }
        } // j
        } // i

//%%%
```

```
/*????????????????????????????????????????????????????????????????????????????????????????????????*/

        //
        //  create output of N symbols of M bits/symbol
        //

        if (block >= B)
        {
            int i;
            int idx = 0;
            int bit = 0;
            unsigned dat = 0x00000000L;
            unsigned prf = 0x00000000L;

            for (i = 0; i < N; i++) {
                while (bit < M) {
                    dat = dat | (rx_trellis_data [idx]<<bit);
                    prf = prf | (rx_trellis_perf [idx]<<bit);
                    bit = bit + U, idx++;
                }
                rx_interleave_data [rx_frame_ptr] [i] = dat & ((1<<M)-1);
                rx_interleave_perf [rx_frame_ptr] [i] = prf & ((1<<M)-1);
                dat = dat >> M;
                prf = prf >> M;
                bit = bit - M;

            }
            if (++rx_frame_ptr >= D) rx_frame_ptr = 0;
        }

        //
        //  convolutionally interleave D depth N symbols of M bits/symbol
        //

        if (block >= B*(D-1))
        {
            int i;
            static long symbol = 0L;
            int idx = 0;
            int frm = rx_frame_ptr;

            for (i = 0; i < N; i++) {
                rx_frame_data [i] = rx_interleave_data [frm] [idx];
                rx_frame_perf [i] = rx_interleave_perf [frm] [idx];
                idx = idx + D;
                if (idx >= N) {
                    idx = idx - N;
                    if (++frm >= D) frm = 0;
                }

                PRINT_POINTS (
                    fprintf (spcout, "%8ld rx_interleave_data %04x\n", symbol, rx_in
terleave_data [rx_frame_ptr] [i]);
                    fprintf (spcout, "%8ld rx_interleave_perf %04x\n", symbol, rx_in
terleave_data [rx_frame_ptr] [i]);
                    fprintf (spcout, "%8ld rx_frame_data %04x\n", symbol, rx_frame_d
ata [i]);
                    fprintf (spcout, "%8ld rx_frame_perf %04x\n", symbol, rx_frame_p
erf [i]);
                    symbol++;
                )
            }
        }

        //
        //  reed-solomon decode frame of N symbols of M bits/symbol
        //

        if (block >= B*(D-1))
        {
            int i;
            int error;
            int total;
            int blind;
            int found;

            for (i = 0; i < NN; i++)
                rs_data [i] = 0;

            for (i = 0; i < K; i++)
                rs_data [i] = rx_frame_data [i];

            for (i = 0; i < R; i++)
                rs_data [KK+i] = rx_frame_data [K+i];

            if (!enable_reed_solomon)   error = -2;
            else                        error = decode_reedsolo (rs_data, NULL, 0);

            for (i = 0; i < K; i++)
                rx_payload_data [i] = rs_data [i];

            for (i = 0; i < K; i++)
                rx_payload_perf [i] = rx_frame_perf [i];

            total = 0;
            blind = 0;
            found = 0;

            for (i = 0; i < N; i++) {
                if (rx_frame_data [i] != rx_frame_perf [i]) total++;
              if (i < K) {
                if (rx_frame_data [i] != rx_frame_perf [i]) blind++;
                if (rx_payload_data [i] != rx_payload_perf [i]) found++;
              }
            }

            PRINT_REED_SOLO (
                fprintf (spcout, "%8ld : %8d %8d %8d %8d\n", block, error, total, bl
ind, found);
            )

//%%%
```

```
        //  descramble payload of K symbols of H bits/symbol
        //

        if (block >= B+(D-1))
        {
            int i;
            static long symbol = 0L;

            for (i = 0; i < K; i++) {
                if (!enable_scrambler)  rx_data [i] = rx_payload_data [i];
                else                    rx_data [i] = scrambler_decode0 (rx_payload_
data [i]);

                if (!enable_scrambler)  rx_perf [i] = rx_payload_perf [i];
                else                    rx_perf [i] = scrambler_decode1 (rx_payload_
perf [i]);

                PRINT_POINTS (
                    fprintf (spcout, "%8ld rx_payload_data %04x\n", symbol, rx_paylo
ad_data [i]);
                    fprintf (spcout, "%8ld rx_payload_perf %04x\n", symbol, rx_paylo
ad_perf [i]);
                    fprintf (spcout, "%8ld rx_data %04x\n", symbol, rx_data [i]);
                    fprintf (spcout, "%8ld rx_perf %04x\n", symbol, rx_perf [i]);
                    symbol++;
                )
            }
        }

        //
        //  check symbols of received and perfect data
        //

        if (block >= B+(D-1))
        {
            int i;
            int error_count = 0;

            for (i = 0; i < K; i++) {
                if (rx_data [i] != rx_perf [i]) error_count++;
            }

            total_error_blocks += (error_count == 0) ? 0 : 1;
            total_count_blocks += 1;

            total_error_symbols += error_count;
            total_count_symbols += K;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//% fprintf (spcout, "%8ld %10.2f - %10.2f %10.2f\n", block+1, -dB (SIGMA*SIGMA), dB
 (total_error_blocks /(double) total_count_blocks)/10.0, dB (total_error_symbols/(dou
ble)total_count_symbols)/10.0);
//% fprintf (stderr, "%8ld %10.2f - %10.2f %10.2f\n", block+1, -dB (SIGMA*SIGMA), dB
 (total_error_blocks /(double) total_count_blocks)/10.0, dB (total_error_symbols/(dou
ble)total_count_symbols)/10.0);
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        )

        if (block >= B+(D-1))
        {
            int i;
            int terminate=0;

            for (i = 0; i < 6; i++) {
                if ((total_count_blocks > count_limit [i]) && (total_error_blocks >
error_limit [i])) {
                    fprintf (spcout, "termination: %8ld>%8ld && %8ld>%8ld\n", total_
count_blocks, count_limit [i], total_error_blocks, error_limit [i]);
                    terminate=1;
                    break;
                }
            }
            if (terminate != 0)
                break;
        }

        if (block >= B+(D-1))
        {
            if ((block%REPORT_LENGTH) == (REPORT_LENGTH-1)) {
            //? fprintf (stderr, "%8ld %10.2f - %8ld %8ld %8ld %8ld\n", block+1, -dB
 (SIGMA*SIGMA), total_error_blocks, total_count_blocks, total_error_symbols, total_co
unt_symbols);
                fprintf (stderr, "%8ld %10.2f - %10.2f %10.2f\n", block+1, -dB (SIGM
A*SIGMA), dB (total_error_blocks /(double) total_count_blocks)/10.0, dB (total_error_
symbols/(double)total_count_symbols)/10.0);
            }
        }
    }
    fprintf (spcout, "%8ld %10.2f - %10.2f %10.2f\n", block, -dB (SIGMA*SIGMA), dB (
total_error_blocks /(double) total_count_blocks)/10.0, dB (total_error_symbols/(doubl
e)total_count_symbols)/10.0);
    fprintf (stderr, "%8ld %10.2f - %10.2f %10.2f\n", block, -dB (SIGMA*SIGMA), dB (
total_error_blocks /(double) total_count_blocks)/10.0, dB (total_error_symbols/(doubl
e)total_count_symbols)/10.0);

    fflush (spcout);
    fflush (stderr);
}

    exit (0);
}
```


WO 00/07323 PCT/US99/17369

```
#ifndef TEST_H
#define TEST_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

#ifdef  WE_ARE_DEFINING_VARIABLES
#define EXTERN
#else   WE_ARE_DEFINING_VARIABLES
#define EXTERN extern
#endif  WE_ARE_DEFINING_VARIABLES

//  define constants

#define MM  8               //  size of basic frame symbol
#define NN  ((1<<MM)-1)     //  maximum size of a frame
#define DD  32              //  maximum depth of interleaver
#define UU  19              //  maximum symbol size
#define BB  5               //  maximum block delay depth
#define LL  50              //  maximum lookback length
#define TT  680             //  maximum trellis symbols per frame

EXTERN int enable_scrambler;
EXTERN int enable_reed_solomon;
EXTERN int enable_interleaver;
EXTERN int enable_trellis_decoder;
EXTERN int enable_turbo_decoder;

EXTERN int print_internal_points;
EXTERN int print_reed_solomon_data;
EXTERN int show_energy_measurements;

EXTERN long BLOCK_LENGTH;
EXTERN long ENERGY_LENGTH;
EXTERN long REPORT_LENGTH;

EXTERN double DB_START;
EXTERN double DB_STOP;
EXTERN double DB_STEP;

EXTERN long count_limit (6);
EXTERN long error_limit (6);

EXTERN double SIGMA;
EXTERN int TIMES;

EXTERN int REED_SOLOMON_SYMBOL_SIZE;
EXTERN int REED_SOLOMON_FRAME_SIZE;
EXTERN int REED_SOLOMON_CHECK_SYMBOLS;
EXTERN int INTERLEAVER_FRAME_DEPTH;
EXTERN int TRELLIS_SYMBOL_SIZE;
EXTERN int TRELLIS_FRAME_SIZE;
EXTERN int TRELLIS_LOOKBACK_LENGTH;
EXTERN int TRELLIS_BLOCK_DELAY;

EXTERN int TURBO_FRAME_SIZE;
EXTERN int TURBO_ITERATION_COUNT;

////// int MM;
////// int NN;
EXTERN int RR;
EXTERN int KK;

EXTERN int M;
EXTERN int N;
EXTERN int R;
EXTERN int K;

////// int DD;
EXTERN int D;

////// int UU;
////// int TT;
////// int LL;
////// int BB;

EXTERN int U;
EXTERN int T;
EXTERN int L;
EXTERN int B;

EXTERN double SCALE;

EXTERN FILE *spcout;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

extern void init_random (double sigma, int times);
extern double random_value (void);

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

extern void init_scrambler (void);
extern unsigned scrambler_encode (unsigned input);
extern unsigned scrambler_decode0 (unsigned input);
extern unsigned scrambler_decode1 (unsigned input);

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

extern void alloc_turbo_memory (unsigned length, unsigned iterations);
extern void free_turbo_memory (void);

extern void init_turbo (double sigma);
extern void turbo_encode (void);
extern void turbo_decode (int blind_decode);

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

extern void init_trellis (void);
extern void trellis_encode (void);
extern void trellis_decode (void);
extern void blind_decode (void);
```


WO 00/07323 PCT/US99/17369

```
EXTERN int tx_Un;
EXTERN int rx_Un;

EXTERN double tx_vx;
EXTERN double tx_vy;
EXTERN double tx_wx;
EXTERN double tx_wy;

EXTERN double rx_vx;
EXTERN double rx_vy;
EXTERN double rx_wx;
EXTERN double rx_wy;

EXTERN double ln_vx;
EXTERN double ln_vy;
EXTERN double ln_wx;
EXTERN double ln_wy;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

EXTERN int tx_bits;
EXTERN int dl_bits;
EXTERN int rx_bits;

EXTERN double tx_A;
EXTERN double tx_B;
EXTERN double tx_C;

EXTERN double ln_A;
EXTERN double ln_B;
EXTERN double ln_C;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

#if (MM <= 8)
typedef unsigned char dtype;
#else
typedef unsigned int dtype;
#endif

void init_rs (void);

void generate_gf (void);     /* Generate Galois Field */
void gen_poly (void);        /* Generate generator polynomial */

int encode_rs (dtype data [], dtype bb []);
int eras_dec_rs (dtype data [], int eras_pos [], int no_eras);

/*
 * Global definitions for Reed-Solomon encoder/decoder
 * Phil Karn KA9Q, September 1996
 *
 * The parameters MM and KK specify the Reed-Solomon code parameters.
 *
 * Set MM to be the size of each code symbol in bits. The Reed-Solomon
 * block size will then be NN = 2**M - 1 symbols. Supported values are
 * defined in rs.c.
 *
 * Set KK to be the number of data symbols in each block, which must be
 * less than the block size. The code will then be able to correct up
 * to NN-KK erasures or (NN-KK)/2 errors, or combinations thereof with
 * each error counting as two erasures.
 */

/*
 * Initialization function
 */

/*
 * These two functions *must* be called in this order (e.g.,
 * by init_rs()) before any encoding/decoding
 */

/*
 * Reed-Solomon encoding
 * data [] is the input block, parity symbols are placed in bb []
 * bb [] may lie past the end of the data, e.g., for (255,223):
 * encode_rs (&data [0], &data [223]);
 */

/*
 * Reed-Solomon erasures-and-errors decoding
 * The received block goes into data[], and a list of zero-origin
 * erasure positions, if any, goes in eras_pos[] with a count in no_eras.
 *
 * The decoder corrects the symbols in place, if possible and returns
 * the number of corrected symbols. If the codeword is illegal or
 * uncorrectible, the data array is unchanged and -1 is returned
 */

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

extern void initialize_options (void);

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

#endif  TEST_H
```

```
#include "test.h"

/* ************************************************************************ */
/* ************************************************************************ */

static int MM_ = MM;
static int NN_ = NN;
static int DD_ = DD;
static int UU_ = UU;
static int II_ = II;
static int LL_ = LL;
static int BB_ = BB;

/* ************************************************************************ */
/* ************************************************************************ */

typedef struct input_s {
    int type;
    char *name;
    void *addr;
} input_t;

#define Null    0
#define Int     1
#define Long    2
#define float   3
#define Format  4
#define Int_    5
#define Long_   6
#define Float_  7

#define crlf    { Format, "", NULL },

static input_t input [] = {

    crlf
    { Int       ,"enable_scrambler"             ,(void *) &enable_scrambler
},
    { Int       ,"enable_reed_solomon"          ,(void *) &enable_reed_solomon
},
    { Int       ,"enable_interleaver"           ,(void *) &enable_interleaver
},
    { Int       ,"enable_trellis_decoder"       ,(void *) &enable_trellis_decoder
},
    { Int       ,"enable_turbo_decoder"         ,(void *) &enable_turbo_decoder ),
    crlf
    { Int       ,"print_internal_points"        ,(void *) &print_internal_points    )
    { Int       ,"print_reed_solomon_data"      ,(void *) &print_reed_solomon_data  )
    { Int       ,"show_energy_measurements"     ,(void *) &show_energy_measurements )
    crlf
    { Long      ,"BLOCK_LENGTH"                 ,(void *) &BLOCK_LENGTH             )
    { Long      ,"ENERGY_LENGTH"                ,(void *) &ENERGY_LENGTH            )
    { Long      ,"REPORT_LENGTH"                ,(void *) &REPORT_LENGTH            )
    crlf
    { Float     ,"DB_START"                     ,(void *) &DB_START                 ),
    { Float     ,"DB_STOP"                      ,(void *) &DB_STOP                  ),
    { Float     ,"DB_STEP"                      ,(void *) &DB_STEP                  ),
    crlf
    { Long      ,"count_limit0"                 ,(void *) &count_limit [0]          ),
    { Long      ,"count_limit1"                 ,(void *) &count_limit [1]          ),
    { Long      ,"count_limit2"                 ,(void *) &count_limit [2]          ),
    { Long      ,"count_limit3"                 ,(void *) &count_limit [3]          ),
    { Long      ,"count_limit4"                 ,(void *) &count_limit [4]          ),
    { Long      ,"count_limit5"                 ,(void *) &count_limit [5]          ),
    crlf
    { Long      ,"error_limit0"                 ,(void *) &error_limit [0]          ),
    { Long      ,"error_limit1"                 ,(void *) &error_limit [1]          ),
    { Long      ,"error_limit2"                 ,(void *) &error_limit [2]          ),
    { Long      ,"error_limit3"                 ,(void *) &error_limit [3]          ),
    { Long      ,"error_limit4"                 ,(void *) &error_limit [4]          ),
    { Long      ,"error_limit5"                 ,(void *) &error_limit [5]          ),
    crlf
////{ Float     ,"SIGMA"                        ,(void *) &SIGMA                    ),
    { Int       ,"TIMES"                        ,(void *) &TIMES                    ),
    crlf
    { Int_      ,"REED_SOLOMON_SYMBOL_SIZE"     ,(void *) &REED_SOLOMON_SYMBOL_SIZE ),
    { Int       ,"REED_SOLOMON_FRAME_SIZE"      ,(void *) &REED_SOLOMON_FRAME_SIZE  ),
    { Int       ,"REED_SOLOMON_CHECK_SYMBOLS"   ,(void *) &REED_SOLOMON_CHECK_SYMBOL
S   ),
    crlf
    { Int       ,"INTERLEAVER_FRAME_DEPTH"      ,(void *) &INTERLEAVER_FRAME_DEPTH  ),
    crlf
    { Int       ,"TRELLIS_SYMBOL_SIZE"          ,(void *) &TRELLIS_SYMBOL_SIZE      ),
    { Int_      ,"TRELLIS_FRAME_SIZE"           ,(void *) &TRELLIS_FRAME_SIZE       ),
    { Int       ,"TRELLIS_LOOKBACK_LENGTH"      ,(void *) &TRELLIS_LOOKBACK_LENGTH  ),
    { Int       ,"TRELLIS_BLOCK_DELAY"          ,(void *) &TRELLIS_BLOCK_DELAY      ),
    crlf
    { Int       ,"TURBO_FRAME_SIZE"             ,(void *) &TURBO_FRAME_SIZE         ),
    { Int       ,"TURBO_ITERATION_COUNT"        ,(void *) &TURBO_ITERATION_COUNT    ),
```


WO 00/07323 PCT/US99/17369

```
    crlf
    { Int_      ,"MM"                   ,(void *) &MM_          }
    { Int_      ,"NN"                   ,(void *) &NN_          }
    { Int_      ,"KK"                   ,(void *) &KK_          }
    { Int_      ,"RR"                   ,(void *) &RR           }
    crlf
    { Int       ,"M"                    ,(void *) &M            }
    { Int       ,"N"                    ,(void *) &N            }
    { Int       ,"K"                    ,(void *) &K            }
    { Int       ,"R"                    ,(void *) &R            }
    crlf
    { Int_      ,"DD"                   ,(void *) &DD_          }
    { Int_      ,"D"                    ,(void *) &D            }
    crlf
    { Int_      ,"UU"                   ,(void *) &UU_          }
    { Int_      ,"TT"                   ,(void *) &TT_          }
    { Int_      ,"LL"                   ,(void *) &LL_          }
    { Int_      ,"BB"                   ,(void *) &BB_          }
    crlf
    { Int       ,"U"                    ,(void *) &U            }
    { Int       ,"T"                    ,(void *) &T            }
    { Int_      ,"L"                    ,(void *) &L            }
    { Int_      ,"B"                    ,(void *) &B            }
    crlf
    { Float     ,"SCALE"                ,(void *) &SCALE        }
    crlf
    crlf
    { Null, NULL, NULL }
};

/* ************************************************************************ */
/* ************************************************************************ */

#define SIZE    1024
char s [SIZE];

static void
parse_input (FILE *fin)
{
    int i;
    int   type;
    char *name;
    int   len;
    void *addr;

    char *t;

    while (fgets (s, SIZE, fin) == s) {

        //  change tabs into spaces
        //  end string if control character found
        //  end string if comment character found

        for (t = s; t [0] != '\0'; t++) {
            if (t [0] == '\t')  t [0] = ' ';
            if (t [0] <  ' ')   break;
            if (t [0] == '#')   break;
        }
        t [0] = '\0';

        //  remove leading white space

        for (t = s; t [0] == ' '; t++);

        //  if no variable name, continue

        if (t [0] == '\0') continue;

        //  search through all variable names
        //  looking for one that matches the input
        //  if not found, print error, exit

        for (i = 0; input [i].type != Null; i++) {
            if (input [i].type == Format) continue;
            if (input [i].type == Int_  ) continue;
            if (input [i].type == Long_ ) continue;
            if (input [i].type == Float_) continue;
            if (input [i].name == NULL) continue;
            if (input [i].addr == NULL) continue;

            type = input [i].type;
            name = input [i].name;
            addr = input [i].addr;
            len  = strlen (name);
            if (strncmp (t, name, len) == 0)
                break;
        }
        if (input [i].type == Null) {
            fprintf (stderr, "error: parse: %s\n", s);
            exit (1);
        }

        //  skip all training white space and punctuation

        for (t += len; t [0] != '\0'; t++) {
            if (t [0] == ' ') continue;
            if (t [0] == '=') continue;
            break;
        }
        if (t [0] == '\0') {
            fprintf (stderr, "error: value: %s\n", s);
            exit (1);
        }
```


WO 00/07323 PCT/US99/17369

```
        // convert the variable value

        switch (type) {
        case Null:  break;
        case Format:break;
        case Int:   *((int    *) addr) = atoi (t);  break;
        case Long:  *((long   *) addr) = atol (t);  break;
        case Float: *((double *) addr) = atof (t);  break;
        default:    break;
        }
    }
}

static void
print_input (FILE *fout)
{
    int i;
    int   type;
    char *name;
    int   len;
    void *addr;

    for (i = 0; input [i].type != Null; i++) {
        type = input [i].type;
        name = input [i].name;
        addr = input [i].addr;

        switch (type) {
        case Null:   break;
        case Format: fprintf (fout, "%s\n", name); break;
        case Int:
        case Int_:   fprintf (fout, "   %-30s   %10d\n",   name, *((int    *) addr))
; break;
        case Long:
        case Long_:  fprintf (fout, "   %-30s   %10ldL\n", name, *((long   *) addr))
; break;
        case Float:
        case Float_: fprintf (fout, "   %-30s   %10.5f\n", name, *((double *) addr))
; break;
        }
    }
}

/* ********************************************************************** */
/* ********************************************************************** */

#define undefined  0

void
initialize_options (void)
{
    int i;

    //
    //  choose basic operations
    //

    enable_scrambler            = 1;
    enable_reed_solomon         = 1;
    enable_interleaver          = 1;
    enable_trellis_decoder      = 1;
    enable_turbo_decoder        = 0;

    print_internal_points       = 1;
    print_reed_solomon_data     = 1;
    show_energy_measurements    = 1;

    //
    //  choose basic simulation
    //

    BLOCK_LENGTH                = 100L;
    ENERGY_LENGTH               = 10000L;
    REPORT_LENGTH               = 10L;

    DB_START                    =  0.0;
    DB_STOP                     = 16.0;
    DB_STEP                     =  1.0;

    count_limit [0]             =       0L;   error_limit [0] = 1000L;
    count_limit [1]             =  500000L;   error_limit [1] =  100L;
    count_limit [2]             = 1000000L;   error_limit [2] =   10L;
    count_limit [3]             = 5000000L;   error_limit [3] =    5L;
    count_limit [4]             = 7500000L;   error_limit [4] =    3L;
    count_limit [5]             = 10000000L;  error_limit [5] =    0L;

    //
    //  choose random number generator
    //

////SIGMA                       = 0.0;
    TIMES                       = 12;

    //
    //  choose scrambler
    //

    //
    //  choose reed solomon
    //

    REED_SOLOMON_SYMBOL_SIZE    = MM;
    REED_SOLOMON_FRAME_SIZE     = 51;
    REED_SOLOMON_CHECK_SYMBOLS  = 16;

    //
    //  choose interleaver
    //

    INTERLEAVER_FRAME_DEPTH     = 4;

    //
    //  choose trellis
    //

    TRELLIS_SYMBOL_SIZE         = 3;
    TRELLIS_LOOKBACK_LENGTH     = 20;
    TRELLIS_BLOCK_DELAY         = 1;

    TURBO_FRAME_SIZE            = 128;
    TURBO_ITERATION_COUNT       = 10;
```


WO 00/07323 PCT/US99/17369

```
    //
    //  parse input files
    //
    ////////////////////////////////////////////////////////////////////////

    parse_input (stdin);

    ////////////////////////////////////////////////////////////////////////
    //
    //  enforce any programming considerations
    //

#define minimize(x,y)       x=(((x)<(y))?(x):(y))
#define maximize(x,y)       x=(((x)>(y))?(x):(y))
#define rangerize(x,y,z)    x=(((x)<(z))?(((x)>(y))?(x):(y)):(z))

#define rangerize(x,y,z)    x=(((x)<(z))?(((x)>(y))?(x):(y)):(z))

    rangerize (enable_scrambler             ,0 ,1);
    rangerize (enable_reed_solomon          ,0 ,1);
    rangerize (enable_interleaver           ,0 ,1);
    rangerize (enable_trellis_decoder       ,0 ,1);
    rangerize (enable_turbo_decoder         ,0 ,1);

    rangerize (print_internal_points        ,0 ,1);
    rangerize (print_reed_solomon_data      ,0 ,1);
    rangerize (show_energy_measurements     ,0 ,1);

    //
    //  choose basic simulation
    //

    rangerize (BLOCK_LENGTH                 ,1L ,1000000000L);
    rangerize (ENERGY_LENGTH                ,1L ,1000000000L);
    rangerize (REPORT_LENGTH                ,1L ,1000000000L);

    rangerize (DB_START                     ,-20.0 ,+20.0);
    rangerize (DB_STOP                      ,-20.0 ,+20.0);
    rangerize (DB_STEP                      ,-20.0 ,+20.0);

    rangerize (count_limit [0]              ,0L ,1000000000L);
    rangerize (count_limit [1]              ,0L ,1000000000L);
    rangerize (count_limit [2]              ,0L ,1000000000L);
    rangerize (count_limit [3]              ,0L ,1000000000L);
    rangerize (count_limit [4]              ,0L ,1000000000L);
    rangerize (count_limit [5]              ,0L ,1000000000L);

    rangerize (error_limit [0]              ,0L ,1000000000L);
    rangerize (error_limit [1]              ,0L ,1000000000L);
    rangerize (error_limit [2]              ,0L ,1000000000L);
    rangerize (error_limit [3]              ,0L ,1000000000L);
    rangerize (error_limit [4]              ,0L ,1000000000L);
    rangerize (error_limit [5]              ,0L ,1000000000L);

////rangerize (SIGMA                        ,0. ,1.);
    rangerize (TIMES                        ,1  ,24);

    ////////////////////////////////////////////////////////////////////////

    if (!enable_scrambler)          ;
    if (!enable_reed_solomon)       REED_SOLOMON_CHECK_SYMBOLS = 0;
    if (!enable_interleaver)        INTERLEAVER_FRAME_DEPTH    = 0;
    if (!enable_trellis_decoder)    TRELLIS_LOOKBACK_LENGTH    = 0;
    if (!enable_turbo_decoder)      ;

    ////////////////////////////////////////////////////////////////////////

    rangerize (REED_SOLOMON_SYMBOL_SIZE     ,MM ,MM);
    rangerize (REED_SOLOMON_FRAME_SIZE      ,0  ,NN);
    rangerize (REED_SOLOMON_CHECK_SYMBOLS   ,0  ,NN);
    rangerize (INTERLEAVER_FRAME_DEPTH      ,1  ,DD);
    rangerize (TRELLIS_SYMBOL_SIZE          ,0  ,UU);
    rangerize (TRELLIS_LOOKBACK_LENGTH      ,0  ,LL);
    rangerize (TRELLIS_BLOCK_DELAY          ,0  ,BB);
    rangerize (TURBO_FRAME_SIZE             ,16 ,1024);
    rangerize (TURBO_ITERATION_COUNT        ,1  ,20);

    TRELLIS_FRAME_SIZE = floor ((double)REED_SOLOMON_SYMBOL_SIZE*(double)REED_SOLOMO
N_FRAME_SIZE/TRELLIS_SYMBOL_SIZE);
    rangerize (TRELLIS_FRAME_SIZE           ,0  ,TT);

    ////////////////////////////////////////////////////////////////////////

////MM = MM;
////NN = NN;
    RR = REED_SOLOMON_CHECK_SYMBOLS;
    KK = NN - RR;

    M  = REED_SOLOMON_SYMBOL_SIZE;
    N  = REED_SOLOMON_FRAME_SIZE;
    R  = REED_SOLOMON_CHECK_SYMBOLS;
    K  = N - R;

////DD = DD;
    D  = INTERLEAVER_FRAME_DEPTH;

////UU = UU;
////TT = TT;
////LL = LL;
////BB = BB;

    U  = TRELLIS_SYMBOL_SIZE;
    T  = TRELLIS_FRAME_SIZE;
    L  = TRELLIS_LOOKBACK_LENGTH;
    B  = TRELLIS_BLOCK_DELAY;

    ////////////////////////////////////////////////////////////////////////

    {
        if (N == 0) {
            fprintf (stderr, "error, too small frame size\n");
            exit (1);
        }
    }

    {
        if (R >= N) {
            fprintf (stderr, "error, too many reed solomon sums requested\n");
            exit (1);
        }
    }
```

5-04 99 7:06p y:PARSE.CPP : 5 of 15

```
{
    if ((N%2) == 0) {
        fprintf (stderr, "error, REED_SOLOMON_FRAME_SIZE should be odd for the i
nterleaver\n");
        exit (1);
    }
}
{
    int value;

    value = pow (2.0, floor (log ((double) INTERLEAVER_FRAME_DEPTH)/log (2.) + 0
.5));

    if (value != INTERLEAVER_FRAME_DEPTH) {
        fprintf (stderr, "error, INTERLEAVER_FRAME_DEPTH should be a power of 2\
n");
        exit (1);
    }
}
{
    int value;

    if (U < 3) {
        fprintf (stderr, "error, IRELLIS_SYMBOL_SIZE must be greater/equal to 3\
n");
        exit (1);
    }

    value = 3*4^(floor((U-3)/4.));

    if (value != U) {
        fprintf (stderr, "error, IRELLIS_SYMBOL_SIZE should satisfy 4i+3 where i
 integer\n");
        exit (1);
    }

    if (T != (double)M*(double)N/U) {
        fprintf (stderr, "error, REED_SOLOMON_FRAME_SIZE should be multiple of IR
ELLIS_SYMBOL_SIZE\n");
        exit (1);
    }
}
{
    if (B*T < L) {
        fprintf (stderr, "error, IRELLIS_BLOCK_DELAY*T should be greater/equal t
o IRELLIS_LOOKBACK_SIZE\n");
    }

    ////////////////////////////////////////////////////////////////////////////

    SCALE = 1.0;;

    ////////////////////////////////////////////////////////////////////////////
#undef  minimize
#undef  maximize
```

```
#undef  rangerize

    ////////////////////////////////////////////////////////////////////////////

    print_input (stderr);
    print_input (spcout);
}
```

5-04-99 4:38p y:\RANDOM.CPP : 1 of 1

```
#include "test.h"

static double SCALE;
static int TIMES;

void
init_random (double sigma, int times)
{
   // srand should have been called within the main routine

   TIMES = times;
   SCALE = sigma * sqrt (12.0) / sqrt ((double) TIMES);
}

double
random_value (void)
{
   int i;
   double result;

   result = 0.0;
   for (i = 0; i < TIMES; i++)
      result += (rand ()/(double) RAND_MAX)-0.5;

   return SCALE * result;
}
```

```
#include "test.h"

/*
 *  Reed-Solomon coding and decoding
 *  Phil Karn (karn@ka9q.ampr.org) September 1996
 *
 *  This file is derived from the program "new_rs_erasures.c" by Robert
 *  Morelos-Zaragoza (robert@spectra.eng.hawaii.edu) and Hari Thirumoorthy
 *  (harit@spectra.eng.hawaii.edu), Aug 1995
 *
 *  I've made changes to improve performance, clean up the code and make it
 *  easier to follow. Data is now passed to the encoding and decoding functions
 *  through arguments rather than in global arrays. The decode function returns
 *  the number of corrected symbols, or -1 if the word is uncorrectable.
 *
 *  This code supports a symbol size from 2 bits up to 16 bits,
 *  implying a block size of 3 2-bit symbols (6 bits) up to 65535
 *  16-bit symbols (1,048,560 bits). The code parameters are set in rs.h.
 *
 *  Note that if symbols larger than 8 bits are used, the type of each
 *  data array element switches from unsigned char to unsigned int. The
 *  caller must ensure that elements larger than the symbol range are
 *  not passed to the encoder or decoder.
 *
 */

#define allocNN NN
#define allocRR NN
#define allocKK NN

//fjh   #if (KK >= NN)
//fjh   #error "KK must be less than 2**MM - 1"
//fjh   #endif

/*
 *  This defines the type used to store an element of the Galois Field
 *  used by the code. Make sure this is something larger than a char if
 *  if anything larger than GF (256) is used.
 *
 *  Note: unsigned char will work up to GF (256) but int seems to run
 *  faster on the Pentium.
 */

typedef int gf;

/*
 *  Primitive polynomials - see Lin & Costello, Appendix A,
 *  and Lee & Messerschmitt, p. 453.
 */

/* ************************************************************************ */
/* ************************************************************************ */

#if (MM == 2)                                   /* Admittedly silly */
static int Pp [MM+1] = { 1, 1, 1 };

#elif (MM == 3)                                 /* 1 + x + x^3 */
static int Pp [MM+1] = { 1, 1, 0, 1 };

#elif (MM == 4)                                 /* 1 + x + x^4 */
static int Pp [MM+1] = { 1, 1, 0, 0, 1 };

#elif (MM == 5)                                 /* 1 + x^2 + x^5 */
static int Pp [MM+1] = { 1, 0, 1, 0, 0, 1 };

#elif (MM == 6)                                 /* 1 + x + x^6 */
static int Pp [MM+1] = { 1, 1, 0, 0, 0, 0, 1 };

#elif (MM == 7)                                 /* 1 + x^3 + x^7 */
static int Pp [MM+1] = { 1, 0, 0, 1, 0, 0, 0, 1 };

#elif (MM == 8)                                 /* 1+x^2+x^3+x^4+x^8 */
static int Pp [MM+1] = { 1, 0, 1, 1, 1, 0, 0, 0, 1 };

#elif (MM == 9)                                 /* 1+x^4+x^9 */
static int Pp [MM+1] = { 1, 0, 0, 0, 1, 0, 0, 0, 0, 1 };

#elif (MM == 10)                                /* 1+x^3+x^10 */
static int Pp [MM+1] = { 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1 };

#elif (MM == 11)                                /* 1+x^2+x^11 */
static int Pp [MM+1] = { 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 };

#elif (MM == 12)                                /* 1+x+x^4+x^6+x^12 */
static int Pp [MM+1] = { 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1 };

#elif (MM == 13)                                /* 1+x+x^3+x^4+x^13 */
static int Pp [MM+1] = { 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 };

#elif (MM == 14)                                /* 1+x+x^6+x^10+x^14 */
static int Pp [MM+1] = { 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1 };

#elif (MM == 15)                                /* 1+x+x^15 */
static int Pp [MM+1] = { 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 };

#elif (MM == 16)                                /* 1+x+x^3+x^12+x^16 */
static int Pp [MM+1] = { 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1 };

#else
#error "MM must be in range 2-16"
#endif

/* ************************************************************************ */
/* ************************************************************************ */

/*
 *  Alpha exponent for the first root of the generator polynomial
 */

#define B0  1

/*
 *  index->polynomial form conversion table
 */

static gf Alpha_to [allocNN+1];

/*
 *  Polynomial->index form conversion table
 */
```


WO 00/07323 PCT/US99/17369

```
static gf Index_of [allocNN+1];

/*
 *  No legal value in index form represents zero, so
 *  we need a special value for this purpose
 */

#define A0   (NN)

/*
 *  Generator polynomial g (x)
 *  Degree of g (x) = 2*TT
 *  has roots @**B0, @** (B0+1), ... ,@^ (B0+2*TT-1)
 */

static gf Gg [allocRR+1];

/*
 *  Compute x % NN, where NN is 2**MM - 1,
 *  without a slow divide
 */

//fjh   static inline gf
//fjh   modnn (int x)

static gf
modnn (int x)
{
    while (x >= NN) {
        x -= NN;
        x = (x >> MM) + (x & NN);
    }
    return x;
}

//fjh   min(a,b)     ((a) < (b) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))

#define CLEAR(a,n) (\
    int ci;\
    for (ci= (n)-1;ci >=0;ci--)\
        (a) [ci] = 0;\
    )

#define COPY(a,b,n) (\
    int ci;\
    for (ci= (n)-1;ci >=0;ci--)\
        (a) [ci] = (b) (ci);\
    )
#define COPYDOWN(a,b,n) (\
    int ci;\
    for (ci= (n)-1;ci >=0;ci--)\
        (a) [ci] = (b) [ci];\
    )

void
init_rs (void)
{
    generate_gf ();
    gen_poly ();
}
```

```
/*
 *  generate GF (2**m) from the irreducible polynomial p (X) in p [0]..p [m]
 *
 *  lookup tables:
 *
 *      index form -> polynomial form   alpha_to [] contains j=alpha**i;
 *      polynomial form -> index form   index_of [j=alpha**i] = i
 *
 *      alpha=2 is the primitive element of GF (2**m)
 *
 *  HARI's COMMENT: (4/13/94) alpha_to [] can be used as follows:
 *
 *      Let @ represent the primitive element commonly called "alpha" that
 *      is the root of the primitive polynomial p (x). Then in GF (2^m), for any
 *      0 <= i <= 2^m-2,
 *
 *          @^i = a (0) + a (1) @ + a (2) @^2 + ... + a (m-1) @^ (m-1)
 *
 *      where the binary vector (a (0), a (1), a (2), ..., a (m-1)) is the represent
ation
 *      of the integer "alpha_to [i]" with a (0) being the LSB and a (m-1) the MSB.
Thus for
 *      example the polynomial representation of @^5 would be given by the binary
 *      representation of the integer "alpha_to [5]".
 *
 *  Similarily, index_of [] can be used as follows:
 *
 *      As above, let @ represent the primitive element of GF (2^m) that is
 *      the root of the primitive polynomial p (x). In order to find the power
 *      of @ (alpha) that has the polynomial representation
 *
 *          a (0) + a (1) @ + a (2) @^2 + ... + a (m-1) @^ (m-1)
 *
 *      we consider the integer "i" whose binary representation with a (0) being LSB
 *      and a (m-1) MSB is (a (0), a (1), ..., a (m-1)) and locate the entry
 *      "index_of [i]". Now, @^index_of [i] is that element whose polynomial
 *      representation is (a (0), a (1), a (2), ..., a (m-1)).
 *
 *  NOTE:
 *
 *      The element alpha_to [2^m-1] = 0 always signifying that the
 *      representation of "@^infinity" = 0 is (0, 0, 0, ..., 0).
 *
 *      Similarily, the element index_of [0] = A0 always signifying
 *      that the power of alpha which has the polynomial representation
 *      (0, 0, ..., 0) is "infinity".
 */

void
generate_gf (void)
{
   register int i, mask;

   mask = 1;
   Alpha_to [MM] = 0;
   for (i = 0; i < MM; i++) {
      Alpha_to [i] = mask;
      Index_of [Alpha_to [i]] = i;

      /*
```


WO 00/07323 PCT/US99/17369

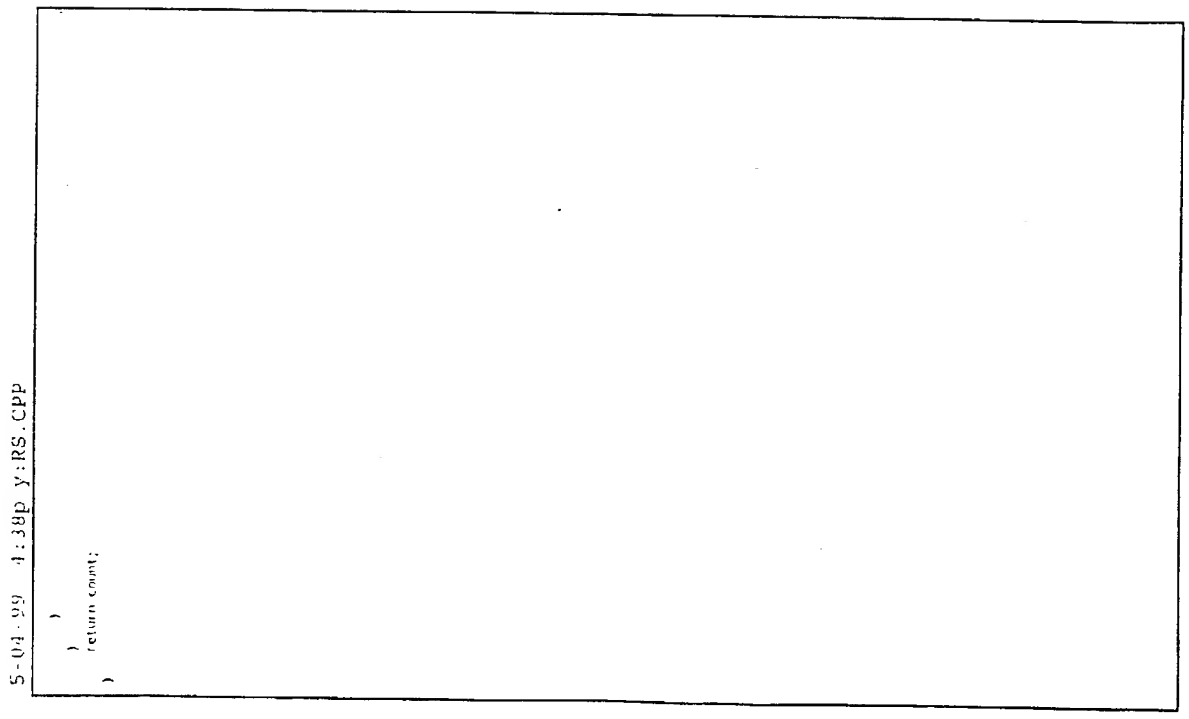
```
         *  If Pp [i] == 1 then, term @^i occurs in poly-repr of @^MM
         */

        if (Pp [i] != 0)
            Alpha_to [MM] ^= mask;

            /*
             *  Bit-wise EXOR operation
             */

        mask <<= 1;

            /*
             *  single left-shift
             */
    }
    Index_of [Alpha_to [MM]] = MM;

    /*
     *  Have obtained poly-repr of @^MM. Poly-repr of @^(i+1) is given by
     *  poly-repr of @^i shifted left one-bit and accounting for any @^MM
     *  term that may occur when poly-repr of @^i is shifted.
     */

    mask >>= 1;
    for (i = MM+1; i < NN; i++) {
        if (Alpha_to [i-1] >= mask)
            Alpha_to [i] = Alpha_to [MM] ^ ((Alpha_to [i-1] ^ mask) << 1);
        else
            Alpha_to [i] = Alpha_to [i-1] << 1;
        Index_of [Alpha_to [i]] = i;
    }
    Index_of [0] = A0;
    Alpha_to [NN] = 0;
}


/*
 *  Obtain the generator polynomial of the TT-error correcting, length
 *  NN= (2**MM -1) Reed Solomon code from the product of (X+@**(B0+i)), i = 0,
 *  ... , (2*TT-1)
 *
 *  Examples:
 *
 *  If B0 = 1, TT = 1. deg (g (x)) = 2*TT = 2.
 *  g (x) = (x+@) (x+@**2)
 *
 *  If B0 = 0, TT = 2. deg (g (x)) = 2*TT = 4.
 *  g (x) = (x+1) (x+@) (x+@**2) (x+@**3)
 */

void
gen_poly (void)
{
    register int i, j;

    Gg [0] = Alpha_to [B0];
    Gg [1] = 1;

        /*
         *  g (x) = (X+@**B0) initially
         */

    for (i = 2; i <= RR; i++) {
        Gg [i] = 1;

        /*
         *  Below multiply (Gg [0]+Gg [1]*x + ... + Gg [i]x^i) by
         *      (@** (B0+i-1) + x)
         */

        for (j = i-1; j > 0; j--) {
            if (Gg [j] != 0)
                Gg [j] = Gg [j-1] ^ Alpha_to [modnn ((Index_of [Gg [j]]) + B0 + i-1)
];
            else
                Gg [j] = Gg [j-1];
        }

        /*
         *  Gg [0] can never be zero
         */

        Gg [0] = Alpha_to [modnn ((Index_of [Gg [0]]) + B0 + i-1)];
    }

    /*
     *  convert Gg [] to index form for quicker encoding
     */

    for (i = 0; i <= RR; i++)
        Gg [i] = Index_of [Gg [i]];
}


/*
 *  take the string of symbols in data [i], i=0.. (k-1) and encode
 *  systematically to produce RR=NN-KK parity symbols in bb [0]..bb [RR-1]
 *  data [] is input and bb [] is output in polynomial form. Encoding is
 *  done by using a feedback shift register with appropriate connections
 *  specified by the elements of Gg [], which was generated above.
 *  Codeword is c (X) = data (X)*X** (RR)+ b (X)
 */

int
encode_rs (dtype data [allocKK], dtype bb [allocRR])
{
    register int i, j;
    gf feedback;

    CLEAR (bb, RR);
    for (i = KK-1; i >= 0; i--) {
#if (MM != 8)
        if (data [i] > NN)
            return -1;

            /*
             *  Illegal symbol
             */
#endif
        feedback = Index_of [data [i] ^ bb [RR-1]];
        if (feedback != A0) {
```

```
            /*
             *  feedback term is non-zero
             */

            for (j = RR-1; j > 0; j--) {
                if (Gg [j] != A0)
                    bb [j] = bb [j-1] ^ Alpha_to [modnn (Gg [j] + feedback)];
                else
                    bb [j] = bb [j-1];
            }
            bb [0] = Alpha_to [modnn (Gg [0] + feedback)];

        } else {

            /*
             *  feedback term is zero. encoder becomes a single-byte shifter
             */

            for (j = RR-1; j > 0; j--)
                bb [j] = bb [j-1];
            bb [0] = 0;
        }
    }
    return 0;
}

/*
 *  Performs ERRORS+ERASURES decoding of RS codes. If decoding is successful,
 *  writes the codeword into data [] itself. Otherwise data [] is unaltered.
 *
 *  Return number of symbols corrected, or -1 if codeword is illegal
 *  or uncorrectable.
 *
 *  First "no_eras" erasures are declared by the calling program. Then, the
 *  maximum # of errors correctable is t_after_eras = floor ((RR-no_eras)/2).
 *  If the number of channel errors is not greater than "t_after_eras" the
 *  transmitted codeword will be recovered. Details of algorithm can be found
 *  in R. Blahut's "Theory ... of Error-Correcting-Codes".
 */

int
eras_dec_rs (dtype data [allocNN], int eras_pos [allocRR], int no_eras)
{
    int deg_lambda, el, deg_omega;
    int i, j, r;
    gf u, q, tmp, num1, num2, den, discr_r;
    static gf recd [allocNN];
    static gf lambda [allocRR+1], s [allocRR+1];

    /*
     *  Err+Eras locator poly and syndrome poly
     */

    static gf b [allocRR+1], t [allocRR+1], omega [allocRR+1];
    static gf root [allocRR], reg [allocRR+1], loc [allocRR];
    int syn_error, count;

    /*
     *  data [] is in polynomial form, copy and convert to index form
     */

    for (i = NN-1; i >= 0; i--) {
#if (MM != 8)
        if (data [i] > NN)
            return -1;

            /*
             *  illegal symbol
             */
#endif
        recd [i] = Index_of [data [i]];
    }

    /*
     *  first form the syndromes; i.e., evaluate recd (x) at roots of
     *  g (x) * namely @** (B0+i), i = 0, ... , (RR-1)
     */

    syn_error = 0;
    for (i = 1; i <= RR; i++) {
        tmp = 0;
        for (j = 0; j < NN; j++) {

            /*
             *  recd [j] in index form
             */

            if (recd [j] != A0)
                tmp ^= Alpha_to [modnn (recd [j] + (B0+i-1)*j)];
        }

        /*
         *  set flag if non-zero syndrome => error
         */

        syn_error |= tmp;

        /*
         *  store syndrome in index form
         */

        s [i] = Index_of [tmp];
    }
    if (!syn_error) {

        /*
         *  if syndrome is zero, data [] is a codeword and there are no
         *  errors to correct. So return data [] unmodified
         */

        return 0;
    }
    CLEAR (&lambda [1], RR);
    lambda [0] = 1;
    if (no_eras > 0) {

        /*
         *  Init lambda to be the erasure locator polynomial
         */

        lambda [1] = Alpha_to [eras_pos [0]];
```

```
        for (i = 1; i < no_eras; i++) {
            u = eras_pos [i];
            for (j = i+1; j > 0; j--) {
                tmp = Index_of [lambda [j-1]];
                if (tmp != A0)
                    lambda [j] ^= Alpha_to [modnn (u + tmp)];
            }
        }

#ifdef ERASURE_DEBUG

        /*
         * find roots of the erasure location polynomial
         */

        for (i=1;i<=no_eras;i++)
            reg [i] = Index_of [lambda [i]];
        count = 0;
        for (i = 1; i <= NN; i++) {
            q = 1;
            for (j = 1; j <= no_eras; j++) {
                if (reg [j] != A0) {
                    reg [j] = modnn (reg [j] + j);
                    q ^= Alpha_to [reg [j]];
                }
            }
            if (!q) {

                /* store root and error location
                 * number indices
                 */

                root [count] = i;
                loc [count] = NN - i;
                count++;
            }
        }
        if (count != no_eras) {
            printf ("\n lambda (x) is WRONG\n");
            return -1;
        }
#ifndef NO_PRINT
        printf ("\n Erasure positions as determined by roots of Eras Loc Poly:\n");
        for (i = 0; i < count; i++)
            printf ("%d ", loc [i]);
        printf ("\n");
#endif
#endif
    }

    for (i=0;i<RR+1;i++)
        b [i] = Index_of [lambda [i]];

    /*
     * Begin Berlekamp-Massey algorithm to determine error+erasure
     * locator polynomial
     */

    r = no_eras;
    el = no_eras;
    while (++r <= RR) {

        /*
         * r is the step number
         */

        /*
         * Compute discrepancy at the r-th step in poly-form
         */

        discr_r = 0;
        for (i = 0; i < r; i++) {
            if ((lambda [i] != 0) && (s [r - i] != A0)) {
                discr_r ^= Alpha_to [modnn (Index_of [lambda [i]] + s [r - i])];
            }
        }
        discr_r = Index_of [discr_r];

            /*
             * Index form
             */

        if (discr_r == A0) {

            /*
             * 2 lines below: B (x) <-- x*B (x)
             */

            COPYDOWN (&b [1], b, RR);
            b [0] = A0;

        } else {

            /*
             * 7 lines below: T (x) <-- lambda (x) - discr_r*x*b (x)
             */

            t [0] = lambda [0];
            for (i = 0 ; i < RR; i++) {
                if (b [i] != A0)
                    t [i+1] = lambda [i+1] ^ Alpha_to [modnn (discr_r + b [i])];
                else
                    t [i+1] = lambda [i+1];
            }

            if (2 * el <= r + no_eras - 1) {
                el = r + no_eras - el;

                /*
                 * 2 lines below: B (x) <-- inv (discr_r) *
                 * lambda (x)
                 */

                for (i = 0; i <= RR; i++)
                    b [i] = (lambda [i] == 0) ? A0 : modnn (Index_of [lambda [i]]
discr_r + NN);

            } else {

                /*
                 * 2 lines below: B (x) <-- x*B (x)
                 */
```

```
                COPYDOWN (&b [1], b, RR);
                b [0] = A0;
            }
            COPY (lambda, t, RR+1);
        }
    }

    /*
     *  Convert lambda to index form and compute deg (lambda (x))
     */

    deg_lambda = 0;
    for (i = 0; i < RR+1; i++) {
        lambda [i] = index_of [lambda [i]];
        if (lambda [i] != A0)
            deg_lambda = i;
    }

    /*
     *  Find roots of the error+erasure locator polynomial. By Chien
     *  Search
     */

    COPY (&reg [1], &lambda [1], RR);
    count = 0;

    /*
     *  Number of roots of lambda (x)
     */

    for (i = 1; i <= NN; i++) {
        q = 1;
        for (j = deg_lambda; j > 0; j--)
            if (reg [j] != A0) {
                reg [j] = modnn (reg [j] + j);
                q ^= Alpha_to [reg [j]];
            }
        if (!q) {

            /*
             *  store root (index-form) and error location number
             */

            root [count] = i;
            loc [count] = NN - i;
            count++;
        }
    }

#ifdef DEBUG
    printf ("\n Final error positions:\t");
    for (i = 0; i < count; i++)
        printf ("%d ", loc [i]);
    printf ("\n");
#endif
    if (deg_lambda != count) {

        /*
         *  deg (lambda) unequal to number of roots => uncorrectable
         *  error detected
         */

        return -1;
    }

    /*
     *  Compute err+eras evaluator poly omega (x) = s (x)*lambda (x) (modulo
     *  x** (RR)). in index form. Also find deg (omega).
     */

    deg_omega = 0;
    for (i = 0; i < RR;i++) {
        tmp = 0;
        j = (deg_lambda < i) ? deg_lambda : i;
        for (;j >= 0; j--) {
            if ((s [i+1 - j] != A0) && (lambda [j] != A0))
                tmp ^= Alpha_to [modnn (s [i+1 - j] + lambda [j])];
        }
        if (tmp != 0)
            deg_omega = i;
        omega [i] = Index_of [tmp];
    }
    omega [RR] = A0;

    /*
     *  Compute error values in poly-form. num1 = omega (inv (X (l))), num2 =
     *  inv (X (l))** (B0-1) and den = lambda_pr (inv (X (l))) all in poly-form
     */

    for (j = count-1; j >=0; j--) {
        num1 = 0;
        for (i = deg_omega; i >= 0; i--) {
            if (omega [i] != A0)
                num1 ^= Alpha_to [modnn (omega [i] + i * root [j])];
        }
        num2 = Alpha_to [modnn (root [j] * (B0-1) + NN)];
        den = 0;

        /*
         *  lambda [i+1] for i even is the formal derivative
         *      lambda_pr of lambda [i]
         */

        for (i = min (deg_lambda, RR-1) & ~1; i >= 0; i -=2) {
            if (lambda [i+1] != A0)
                den ^= Alpha_to [modnn (lambda [i+1] + i * root [j])];
        }
        if (den == 0) {
#ifdef DEBUG
            printf ("\n ERROR: denominator = 0\n");
#endif
            return -1;
        }

        /*
         *  Apply error to data
         */

        if (num1 != 0) {
            data [loc [j]] ^= Alpha_to [modnn (Index_of [num1] + Index_of [num2] + N
N - Index_of [den])];
```


WO 00/07323 PCT/US99/17369


5-04-99 4:38p y:RS.CPP
return count;

```
#include "test.h"

// delay 0 number 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1
7 18 19 20 21 22 23 24
// bit position 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
// byte position ------------lsb------------ ------------nsb------------
------------msb------------
//      76543210  a b c d e f g h i j k l m n o p
q r s t u v w x
//                ------------lsb------------ ------------nsb------------
------------msb------------
//      -7654321  0rw a b c d e f g h i j k l m n o
p q r s t u v w
//      --765432  1qv 0rw a b c d e f g h i j k l m n
o p q r s t u v
//      ---76543  2pu 1qv 0rw a b c d e f g h i j k l m
n o p q r s t u
//      ----7654  3ot 2pu 1qv 0rw a b c d e f g h i j k l
m n o p q r s t
//      -----765  4ns 3ot 2pu 1qv 0rw a b c d e f g h i j k
l m n o p q r s
//      ------76  5mr 4ns 3ot 2pu 1qv 0rw a b c d e f g h i j
k l m n o p q r
//      -------7  6lq 5mr 4ns 3ot 2pu 1qv 0rw a b c d e f g h i
j k l m n o p q
//      --------  7kp 6lq 5mr 4ns 3ot 2pu 1qv 0rw a b o d e f g h
i j k l m n o p
//                ------------lsb------------ ------------nsb------------
------------msb------------
//                -----------output-----------

void
init_scrambler (void)
{
}

unsigned
scrambler_encode (unsigned input)
{
    unsigned upper;
    unsigned output;
    static unsigned lsb = 0;
    static unsigned nsb = 0;
    static unsigned msb = 0;

    upper  = (nsb << 8) + msb;
    output = input ^ (upper >> 6) ^ (upper >> 1);

    input  &= 0x00ff;
    output &= 0x00ff;

    msb = nsb;
    nsb = lsb;
    lsb = output;

    return output;
}

unsigned
```

```
scrambler_decode0 (unsigned input)
{
    unsigned upper;
    unsigned output;
    static unsigned lsb = 0;
    static unsigned nsb = 0;
    static unsigned msb = 0;

    upper  = (nsb << 8) + msb;
    output = input ^ (upper >> 6) ^ (upper >> 1);

    input  &= 0x00ff;
    output &= 0x00ff;

    msb = nsb;
    nsb = lsb;
    lsb = input;

    return output;
}

unsigned
scrambler_decode1 (unsigned input)
{
    unsigned upper;
    unsigned output;
    static unsigned lsb = 0;
    static unsigned nsb = 0;
    static unsigned msb = 0;

    upper  = (nsb << 8) + msb;
    output = input ^ (upper >> 6) ^ (upper >> 1);

    input  &= 0x00ff;
    output &= 0x00ff;

    msb = nsb;
    nsb = lsb;
    lsb = input;

    return output;
}
```

```
#include "test.h"

#define I0  0x0
#define I1  0x1
#define I2  0x2
#define I3  0x3
#define I4  0x4
#define I5  0x5
#define I6  0x6
#define I7  0x7
#define I8  0x8
#define I9  0x9
#define Ia  0xa
#define Ib  0xb
#define Ic  0xc
#define Id  0xd
#define Ie  0xe
#define If  0xf

#define S0  0x0
#define S1  0x1
#define S2  0x2
#define S3  0x3
#define S4  0x4
#define S5  0x5
#define S6  0x6
#define S7  0x7
#define S8  0x8
#define S9  0x9
#define Sa  0xa
#define Sb  0xb
#define Sc  0xc
#define Sd  0xd
#define Se  0xe
#define Sf  0xf

#define H0  0x0
#define H1  0x1
#define H2  0x2
#define H3  0x3
#define H4  0x4
#define H5  0x5
#define H6  0x6
#define H7  0x7

typedef struct {
    double x;
    double y;
} f2D_t;

typedef struct {
    f2D_t v;
    f2D_t w;
} f4D_t;

typedef struct {
    int x;
    int y;
} p2D_t;

typedef struct {
    p2D_t v;
    p2D_t w;
} p4D_t;

typedef struct {
    p2D_t p;
    double m;
} table2D_t;

typedef struct {
    p4D_t p;
    double m;
} table4D_t;

typedef struct {
    p4D_t p;
    int l;
} tableLB_t;

typedef struct {
    double m;
} scores_t;

static table2D_t table2Dv [4];
static table2D_t table2Dw [4];
static table4D_t table4D  [8];
static tableLB_t tableLB  [LL+1] [16];

static int pointerLB;

static int tx_s [4];
static int tx_u [4], rx_u [4];
static int tx_v [2], rx_v [2];
static int tx_w [2], rx_w [2];

static int    rx_LB;
static double rx_MN;

static p4D_t  tx_p, rx_p;
static f4D_t  tx_f, rx_f, in_f;

static unsigned next_table [16] = {     // u3 u2 u1 u0  to  v1 v0 w1 w0
    /* 0x0 */   0x0,
    /* 0x1 */   0x2,
    /* 0x2 */   0xa,
    /* 0x3 */   0x8,
    /* 0x4 */   0x3,
    /* 0x5 */   0x1,
    /* 0x6 */   0x9,
    /* 0x7 */   0xb,
    /* 0x8 */   0x1,
    /* 0x9 */   0xd,
    /* 0xa */   0x5,
    /* 0xb */   0x7,
    /* 0xc */   0xc,
    /* 0xd */   0xe,
    /* 0xe */   0x6,
    /* 0xf */   0x4,
};
```


WO 00/07323 PCT/US99/17369

```
static unsigned back_table [16] = {     // v1 v0 w1 w0  to  u3 u2 u1 u0
    /* 0x0 */   0x0,
    /* 0x1 */   0x5,
    /* 0x2 */   0x1,
    /* 0x3 */   0x4,
    /* 0x4 */   0xf,
    /* 0x5 */   0xa,
    /* 0x6 */   0xe,
    /* 0x7 */   0xb,
    /* 0x8 */   0x3,
    /* 0x9 */   0x6,
    /* 0xa */   0x2,
    /* 0xb */   0x7,
    /* 0xc */   0xc,
    /* 0xd */   0x9,
    /* 0xe */   0xd,
    /* 0xf */   0x8,
};

static scores_t best_scores [16];
static scores_t  old_scores [16];

static void
create2D (table2D_t table2D [], f2D_t u, p2D_t p)
{
    int i;

    i = 2*((p.x>>1)&1) + 1*((p.y>>1)&1);

//?fprintf (stderr, "%6.2f %6.2f - %4d %4d - %2d", u.x, u.y, p.x, p.y, i);

    table2D [i].p = p;
    table2D [i].m = (u.x-p.x)*(u.x-p.x) + (u.y-p.y)*(u.y-p.y);

//?fprintf (stderr, " = %4d %4d - %10.5f", table2D [i].p.x, table2D [i].p.x, table2D
 [i].m);

//?fprintf (stderr, "\n");
}

static void
build2D (table2D_t table2D [], f2D_t f)
{
    f2D_t u;
    p2D_t pa, pb, pc, pd;

    u.x = f.x / SCALE;
    u.y = f.y / SCALE;

    pa.x = 1 + 2*((u.x<0.0)?-ceil(-u.x/2):+floor(+u.x/2));
    pa.y = 1 + 2*((u.y<0.0)?-ceil(-u.y/2):+floor(+u.y/2));

    pb.x = pa.x + 2*((u.x<pa.x)?-1:+1); pb.y = pa.y;
    pc.x = pa.x;                        pc.y = pa.y + 2*((u.y<pa.y)?-1:+1);
    pd.x = pb.x;                        pd.y = pc.y;

    create2D (table2D, u, pa);
    create2D (table2D, u, pb);
    create2D (table2D, u, pc);
    create2D (table2D, u, pd);
}
```

```
static void
form4D (int u, int v1st, int w1st, int v2nd, int w2nd)
{
    double m;
    int v, w;

    double m1st = table2Dv [v1st].m + table2Dw [w1st].m;
    double m2nd = table2Dv [v2nd].m + table2Dw [w2nd].m;

    if (m1st <= m2nd)   m = m1st, v = v1st, w = w1st;
    else                m = m2nd, v = v2nd, w = w2nd;

    table4D [u].p.v = table2Dv [v].p;
    table4D [u].p.w = table2Dw [w].p;
    table4D [u].m   = m;
}

static void
build4D (void)
{
    form4D (0,  0,0,  3,3);
    form4D (1,  0,2,  3,1);
    form4D (2,  1,1,  2,2);
    form4D (3,  1,3,  2,0);
    form4D (4,  0,3,  3,0);
    form4D (5,  0,1,  3,2);
    form4D (6,  1,2,  2,1);
    form4D (7,  1,0,  2,3);
}

static void
cleanup (void)
{
    int i;
    int    link;
    double minu;
    int p;
    int index;

//?fprintf (stderr, "*%10.5f ", best_scores [0].m);

    old_scores [0].m = best_scores [0].m;
    minu = best_scores [0].m, link = 0;

    for (i = 1; i < 16; i++) {
//?fprintf (stderr, "*%10.5f ", best_scores [i].m);
        old_scores [i].m = best_scores [i].m;
        if (minu > best_scores [i].m)
            minu = best_scores [i].m, link = i;
    }
    rx_HN = minu;
    rx_LB = link;

//?fprintf (stderr, " = %10.5f %4d", rx_HN, rx_LB);
//?fprintf (stderr, "\n");

    p = pointerLB;
    for (i = 0; i < L; i++) {
        link = tableLB [p] [link].l;
        if (--p < 0) p = L;
```

```
    )
    pointerLB = p;

    rx_p = tableLB [pointerLB] [link].p;

    rx_f.v.x = rx_p.v.x * SCALE;
    rx_f.v.y = rx_p.v.y * SCALE;
    rx_f.w.x = rx_p.w.x * SCALE;
    rx_f.w.y = rx_p.w.y * SCALE;

    rx_v[1] = (rx_p.v.x>>1) & 1;
    rx_v[0] = (rx_p.v.y>>1) & 1;
    rx_w[1] = (rx_p.w.x>>1) & 1;
    rx_w[0] = (rx_p.w.y>>1) & 1;

    index = 8*rx_v[1] + 4*rx_v[0] + 2*rx_w[1] + 1*rx_w[0];

    rx_u[3] = (back_table [index] >> 3) & 1;
    rx_u[2] = (back_table [index] >> 2) & 1;
    rx_u[1] = (back_table [index] >> 1) & 1;
    rx_u[0] = (back_table [index] >> 0) & 1;
}

static void
update (int state, int i1st, int i2nd, int i3rd, int i4th, int s1st, int s2nd, int s
3rd, int s4th)
{
    double m1st, m2nd, m3rd, m4th;

    m1st = table4D [i1st].m + old_scores [s1st].m;
    m2nd = table4D [i2nd].m + old_scores [s2nd].m;
    m3rd = table4D [i3rd].m + old_scores [s3rd].m;
    m4th = table4D [i4th].m + old_scores [s4th].m;

//?fprintf (stderr, "%x %10.5f %10.5f %10.5f %10.5f", state, m1st, m2nd, m3rd, m4th)
;

    if (m1st >= m2nd) goto not_1st;
    if (m1st >= m3rd) goto not_1st_2nd;
    if (m1st >= m4th) goto not_1st_2nd_3rd;

            best_scores [state].m = m1st;
        tableLB [pointerLB] [state].l = s1st;
        tableLB [pointerLB] [state].p = table4D [i1st].p;
//?fprintf (stderr, " - m1st\n");
        return;

not_1st:

    if (m2nd >= m3rd) goto not_1st_2nd;
    if (m2nd >= m4th) goto not_1st_2nd_3rd;

            best_scores [state].m = m2nd;
        tableLB [pointerLB] [state].l = s2nd;
        tableLB [pointerLB] [state].p = table4D [i2nd].p;
//?fprintf (stderr, " - m2nd\n");
        return;

not_1st_2nd:

    if (m3rd >= m4th) goto not_1st_2nd_3rd;

            best_scores [state].m = m3rd;
        tableLB [pointerLB] [state].l = s3rd;
        tableLB [pointerLB] [state].p = table4D [i3rd].p;
//?fprintf (stderr, " - m3rd\n");
        return;

not_1st_2nd_3rd:

            best_scores [state].m = m4th;
        tableLB [pointerLB] [state].l = s4th;
        tableLB [pointerLB] [state].p = table4D [i4th].p;
//?fprintf (stderr, " - m4th\n");
        return;
}

void
trellis_decode (void)
{
    ln_f.v.x = ln_vx;                       // wrapper
    ln_f.v.y = ln_vy;                       // wrapper
    ln_f.w.x = ln_wx;                       // wrapper
    ln_f.w.y = ln_wy;                       // wrapper

    build2D (table2Dv, ln_f.v);
    build2D (table2Dw, ln_f.w);
    build4D ();

    update (T0, M0,M4,M2,M6, S0,S4,S8,Sc);
    update (T1, M2,M6,M0,M4, S0,S4,S8,Sc);
    update (T2, M4,M0,M6,M2, S0,S4,S8,Sc);
    update (T3, M6,M2,M4,M0, S0,S4,S8,Sc);
    update (T4, M1,M5,M3,M7, S1,S5,S9,Sd);
    update (T5, M3,M7,M1,M5, S1,S5,S9,Sd);
    update (T6, M5,M1,M7,M3, S1,S5,S9,Sd);
    update (T7, M7,M3,M5,M1, S1,S5,S9,Sd);
    update (T8, M2,M6,M0,M4, S2,S6,Sa,Se);
    update (T9, M0,M4,M2,M6, S2,S6,Sa,Se);
    update (Ta, M6,M2,M4,M0, S2,S6,Sa,Se);
    update (Tb, M4,M0,M6,M2, S2,S6,Sa,Se);
    update (Tc, M3,M7,M1,M5, S3,S7,Sb,Sf);
    update (Td, M1,M5,M3,M7, S3,S7,Sb,Sf);
    update (Te, M7,M3,M5,M1, S3,S7,Sb,Sf);
    update (Tf, M5,M1,M7,M3, S3,S7,Sb,Sf);

    cleanup ();

    rx_vx = rx_f.v.x;                       // wrapper
    rx_vy = rx_f.v.y;                       // wrapper
    rx_wx = rx_f.w.x;                       // wrapper
    rx_wy = rx_f.w.y;                       // wrapper

    rx_Un = 4*rx_u[3] + 2*rx_u[2] + 1*rx_u[1];  // wrapper
}

void
trellis_encode (void)
{
    int index;

    tx_u[3] = (tx_Un>>2)&1;                 // wrapper
```


WO 00/07323 PCT/US99/17369

```
    tx_u[2] = (tx_Un>>1)&1;                    // wrapper
    tx_u[1] = (tx_Un>>0)&1;                    // wrapper

    tx_u[0] = tx_s[0];

    tx_s[0] = tx_s[3]  tx_s[1]  tx_u[1];
    tx_s[3] = tx_s[1]                ;
    tx_s[1] = tx_s[2]           tx_u[2];
    tx_s[2] =                   tx_u[0];

    index = 8*tx_u[3] + 4*tx_u[2] + 2*tx_u[1] + 1*tx_u[0];

    tx_v[1] = (next_table [index] >> 3) & 1;
    tx_v[0] = (next_table [index] >> 2) & 1;
    tx_w[1] = (next_table [index] >> 1) & 1;
    tx_w[0] = (next_table [index] >> 0) & 1;

    tx_p.v.x = (tx_v[1]?-1:+1);
    tx_p.v.y = (tx_v[0]?-1:+1);
    tx_p.w.x = (tx_w[1]?-1:+1);
    tx_p.w.y = (tx_w[0]?-1:+1);

    tx_f.v.x = tx_p.v.x * SCALE;
    tx_f.v.y = tx_p.v.y * SCALE;
    tx_f.w.x = tx_p.w.x * SCALE;
    tx_f.w.y = tx_p.w.y * SCALE;

    tx_vx = tx_f.v.x;                          // wrapper
    tx_vy = tx_f.v.y;                          // wrapper
    tx_wx = tx_f.w.x;                          // wrapper
    tx_wy = tx_f.w.y;                          // wrapper
}

void
init_trellis (void)
{
    int i;
    int p;

    tx_s[0] = 0;
    tx_s[1] = 0;
    tx_s[2] = 0;
    tx_s[3] = 0;

    for (i = 0; i < 16; i++)
        old_scores [i].m = 10.0;

    for (p = 0; p < L+1; p++) {
        for (i = 0; i < 16; i++) {
            tableLB [p] [i].p.v.x = 0;
            tableLB [p] [i].p.v.y = 0;
            tableLB [p] [i].p.w.x = 0;
            tableLB [p] [i].p.w.y = 0;
            tableLB [p] [i].l     = 0;
        }
    }

    old_scores [0].m = 0.0;
    pointerLB = 0;
}
```

```
void
blind_decode (void)
{
    static f4D_t ln_f;
    static f4D_t rx_a;
    static p4D_t rx_p;
    static f4D_t rx_f;
    static int rx_v [2];
    static int rx_w [2];
    static int rx_u [4];
    static int index;

    ln_f.v.x = ln_vx;                          // wrapper
    ln_f.v.y = ln_vy;                          // wrapper
    ln_f.w.x = ln_wx;                          // wrapper
    ln_f.w.y = ln_wy;                          // wrapper

    rx_a.v.x = ln_f.v.x / SCALE;
    rx_a.v.y = ln_f.v.y / SCALE;
    rx_a.w.x = ln_f.w.x / SCALE;
    rx_a.w.y = ln_f.w.y / SCALE;

    rx_p.v.x = 1 + 2*((rx_a.v.x<0.0)?-ceil(-rx_a.v.x/2):+floor(+rx_a.v.x/2));
    rx_p.v.y = 1 + 2*((rx_a.v.y<0.0)?-ceil(-rx_a.v.y/2):+floor(+rx_a.v.y/2));
    rx_p.w.x = 1 + 2*((rx_a.w.x<0.0)?-ceil(-rx_a.w.x/2):+floor(+rx_a.w.x/2));
    rx_p.w.y = 1 + 2*((rx_a.w.y<0.0)?-ceil(-rx_a.w.y/2):+floor(+rx_a.w.y/2));

    rx_v[1] = (rx_p.v.x>>1) & 1;
    rx_v[0] = (rx_p.v.y>>1) & 1;
    rx_w[1] = (rx_p.w.x>>1) & 1;
    rx_w[0] = (rx_p.w.y>>1) & 1;

    index = 8*rx_v[1] + 4*rx_v[0] + 2*rx_w[1] + 1*rx_w[0];

    rx_u[3] = (back_table [index] >> 3) & 1;
    rx_u[2] = (back_table [index] >> 2) & 1;
    rx_u[1] = (back_table [index] >> 1) & 1;
    rx_u[0] = (back_table [index] >> 0) & 1;

    rx_f.v.x = rx_p.v.x * SCALE;
    rx_f.v.y = rx_p.v.y * SCALE;
    rx_f.w.x = rx_p.w.x * SCALE;
    rx_f.w.y = rx_p.w.y * SCALE;

    rx_vx = rx_f.v.x;                          // wrapper
    rx_vy = rx_f.v.y;                          // wrapper
    rx_wx = rx_f.w.x;                          // wrapper
    rx_wy = rx_f.w.y;                          // wrapper

    rx_Un = 4*rx_u[3] + 2*rx_u[2] + 1*rx_u[1];  // wrapper
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```


WO 00/07323 PCT/US99/17369

```
#undef  HEADER_LETTER
#undef  HEADER_VALUES
#undef  HEADER_RETURN
#undef  MINIMUM_NUMERATOR
#undef  MINIMUM_DENOMINATOR

#ifdef  HEADER_LETTER
bool   bHEADER_LETTER = true;
#else   HEADER_LETTER
bool   bHEADER_LETTER = false;
#endif  HEADER_LETTER

#ifdef  HEADER_VALUES
bool   bHEADER_VALUES = true;
#else   HEADER_VALUES
bool   bHEADER_VALUES = false;
#endif  HEADER_VALUES

#ifdef  HEADER_RETURN
bool   bHEADER_RETURN = true;
#else   HEADER_RETURN
bool   bHEADER_RETURN = false;
#endif  HEADER_RETURN

#ifdef  MINIMUM_NUMERATOR
bool   bMINIMUM_NUMERATOR = true;
#else   MINIMUM_NUMERATOR
bool   bMINIMUM_NUMERATOR = false;
#endif  MINIMUM_NUMERATOR

#ifdef  MINIMUM_DENOMINATOR
bool   bMINIMUM_DENOMINATOR = true;
#else   MINIMUM_DENOMINATOR
bool   bMINIMUM_DENOMINATOR = false;
#endif  MINIMUM_DENOMINATOR

////////////////////////////////////////////////////////////////////////////////
//
//  fjh this file was random.h
//

#include <stdlib.h>
#include <stdio.h>
#include <values.h>
#include <math.h>
#include <time.h>
#include <limits.h>

//  fjh #ifndef RANDOM_H
//  fjh #define RANDOM_H

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS MINDOUBLE
#define RNMX (1.0-EPS)

// a uniform random number generator between zero and 1.

class Random
{
    long idum2;
   long idum;
    long iy;
    long iv[NTAB];

   unsigned memory;
   void init(long seed);
   double ran2();
public:
    Random(long seed);
   Random();
    double doublerandom();
   long longrandom(long range);
   bool boolrandom();
};

//  fjh #endif  // RANDOM_H

////////////////////////////////////////////////////////////////////////////////
//
//  fjh this file was random.cpp
//

//  fjh #include "random.h"

/*
Long period (? 2 \Theta 10 18 ) random number generator of L'Ecuyer with Bays|Durham
shuffle
and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusiv
e of
the endpoint values). Call with idum a negative integer to initialize; thereafter, d
o not alter
idum between successive deviates in a sequence. RNMX should approximate the largest
floating
value that is less than 1.
--
*/

double Random::ran2()
{
    int j;
    long k;
    double temp;


    k=(idum)/IQ1;
    idum=IA1*(idum-k*IQ1)-k*IR1;  // Compute idum=(IA1*idum) % IM1 without overflows
 by Schrage's method.
    if (idum < 0)
        idum += IM1;
    k=idum2/IQ2;
```


WO 00/07323 PCT/US99/17369

```
    idum2=IA2*(idum2-k*IQ2)-k*IR2;  // Compute idum2=(IA2*idum) % IM2 likewise.
    if (idum2 < 0)
    idum2 += IM2;
   j = iy/NDIV;
   iy=iv[j]-idum2;
//  iy=iv[j]-idum2;          // Here idum is shuffled, idum and idum2 are combined to
 generate output.
    iv[j] = idum;
    if (iy < 1)
        iy += IMM1;
    if ((temp=AM*iy) > RNMX)
        return RNMX;         // Because users don't expect endpoint values.
    else
        return temp;
}


void Random::init(long seed)
{
    idum2=123456789;
   idum=0;
    iy=0;

    if (seed != 0)
        idum = seed;
   else
    idum = 1;


    for (int j=NTAB+7;j>=0;j--) // Load the shuffle table (after 8 warm-ups).
    {
        long k=(idum)/IQ1;

        idum=IA1*(idum-k*IQ1)-k*IR1;
        if (idum < 0)
            idum += IM1;
        if (j < NTAB)
            iv[j] = idum;
    }
    iy=iv[0];
}

Random::Random(long seed)
{
    init(seed);
}

Random::Random()
{
   time_t t;

   time(&t);

   init((long)t);
}

double Random::doublerandom()
{
    double t = ran2();
   return t;
}
```

```
long Random::longrandom(long range)
{
    double t;

   t = doublerandom();
   return((long)(t*(double)range));
}

bool Random::boolrandom()
{
    double t=doublerandom();

   if (t>0.5)
    return true;
   else
    return false;
}

////////////////////////////////////////////////////////////////////////////
//
//  fjh this file was tf.h
//

extern void createencodetable();
extern void createinterleave(unsigned size);
extern void interleave(bool *mesg,unsigned size);
extern void encode(bool *mesg,bool *parity,unsigned size, bool force);
extern void deinterleave(bool *mesg,unsigned size);
extern double gaussian(double variance);
extern unsigned decode (double *mesg, double *parity1, double *parity2, unsigned siz
e, bool *boolmesg);

extern unsigned *interleavearray;
extern unsigned *deinterleavearray;
extern Random before;

bool boolrand (void) { return before.boolrandom(); }

//  fjh extern unsigned int ITERATIONS;
//  fjh extern unsigned int N;
//  fjh extern double No;

static unsigned int N=128;
static unsigned int ITERATIONS=20;
static double No;

////////////////////////////////////////////////////////////////////////////
//
//  fjh this file was tf.cpp
//

//  fjh #include "random.h"
//  fjh #include "tf.h"

// zero mean RV with variance as given
// encodes mesg into parity1, and if force it true it modifies mesg to
// force the encoder to the zero state by the last bit.
// binary addition with no carry

double gaussian(double variance);
```


WO 00/07323 PCT/US99/17369

```
void encode(bool *mesg,bool *parity1,unsigned size, bool force);
void interleave(bool *mesg, unsigned size);
bool add(bool a,bool b);
void interleave(bool *mesg, unsigned size);
void deinterleave(bool *mesg, unsigned size);
void deinterleavedouble(double *mesg, unsigned size);
void interleavedouble(double *mesg, unsigned size);
void createinterleave(unsigned size);
void createencodetable();
void inttobool(unsigned state, bool *array, unsigned size);
void booltoint(bool *array, unsigned size, unsigned *state);
unsigned decode (double *channelmesg, double *parity1, double *parity2, unsigned siz
e, bool *mesg);

unsigned *interleavearray;
unsigned *deinterleavearray;

// global information about the encoder used by the encoder and the decoder
// routine
// how many states are in the encoder (a power of 2)
// log2(numstates)
// [2] = input, [16] = current state, tostate[2][16] = next state
// [2] = last input, [16] = current state, fromstate[2][16] = previous state
// [2] = input, [16] = current state, output[2][16] = output of encoder

unsigned numstates;
unsigned memory;
unsigned *tostate[2];
unsigned *fromstate[2];
bool *output[2];
Random before;

void createencodetable()
{
    bool *boolstate;
    bool *newstate;

    numstates = 16;
    memory = 4;

    // create arrays used by encode and decode
    output[0] = new bool[numstates];
    output[1] = new bool[numstates];
    fromstate[0] = new unsigned[numstates];
    fromstate[1] = new unsigned[numstates];
    tostate[0] = new unsigned[numstates];
    tostate[1] = new unsigned[numstates];

    boolstate = new bool[memory];
    newstate = new bool[memory];

    for (unsigned input=0;input<2;input++) {
        for (unsigned intstate=0;intstate<numstates;intstate++) {
            bool boolinput = (input == 0) ? false : true;

            inttobool(intstate,boolstate,memory);

            // calculate output due to the input
            output[input][intstate] = add(boolinput,boolstate[0]);
            output[input][intstate] = add(output[input][intstate],boolstate[3]);
            output[input][intstate] = add(output[input][intstate],boolstate[1]);
            output[input][intstate] = add(output[input][intstate],boolstate[2]);
            output[input][intstate] = add(output[input][intstate],boolstate[3]);

            // calculate new states
            newstate[3] = boolstate[2];
            newstate[2] = boolstate[1];
            newstate[1] = boolstate[0];
            newstate[0] = add(add(boolinput,boolstate[0]),boolstate[3]);
            // from s' to s
            booltoint (newstate,memory,&tostate[input][intstate]);
            // from s to s'
            fromstate[input][tostate[input][intstate]] = intstate;
        }
    }

    delete boolstate;
    delete newstate;
}

void inttobool(unsigned state, bool *array, unsigned size)
{
    for (unsigned x=0;x<size;x++)
    {
        unsigned next = state >> 1;

        if ((next << 1) == state)
            array[x] = false;
        else
            array[x] = true;

        state = next;
    }
}

void booltoint(bool *array, unsigned size, unsigned *state)
{
    *state = 0;

    for (int x=0;x<size;x++)
    if (array[x] == true)
        (*state) |= (1 << x);
}

unsigned decode (double *mesg, double *parity1, double *parity2, unsigned size, bool
 *boolmesg)
{
    static double **a[2];
    static double **b[2];
    static double *L[2];
    static double **gamma[2][2];
    static double **gammaE[2][2];
    static bool initialized=false;
    static unsigned lastsize=0;
    unsigned returnvalue=ITERATIONS;

    // minimize new's and delete's to only when needed
    if (size != lastsize && initialized == true)
    {
        // delete all the arrays and rebuild
        for (int y=0;y<2;y++) {
            for (int x=0;x<2;x++) {
```


WO 00/07323 PCT/US99/17369

```
                for (int z=0;z<lastsize;z++) {
                    delete gamma[y][x][z];
                    delete gammaE[y][x][z];
                }
                delete gamma[y][x];
                delete gammaE[y][x];
            }
        }

        // create L[encoder #]
        for (int y=0;y<2;y++)
            delete L[y];

        // create alpha[encoder #][k][state]
        for (int x=0;x<2;x++) {
            for (int y=0;y<lastsize;y++) {
                delete a[x][y];
                delete b[x][y];
            }

            delete a[x];
            delete b[x];
        }
    }

    if (initialized == false || size != lastsize) {
        initialized = true;
        lastsize = size;

        // create the arrays dynamically at runtime, delete at end of routine
        // create gamma[encoder #][uk][k][state]
        for (int y=0;y<2;y++) {
            for (int x=0;x<2;x++) {
                gamma[y][x] = new double*[size];
                gammaE[y][x] = new double*[size];
                for (int z=0;z<size;z++) {
                    gamma[y][x][z] = new double[numstates];
                    gammaE[y][x][z] = new double[numstates];
                }
            }
        }

        // create L[encoder #]
        for (int y=0;y<2;y++)
            L[y] = new double[size];

        // create alpha[encoder #][k][state]
        for (int x=0;x<2;x++) {
            a[x] = new double*[size];
            b[x] = new double*[size];
            // each Yk has 'numstates' values of gamma
            for (int y=0;y<size;y++) {
                a[x][y] = new double[numstates];
                b[x][y] = new double[numstates];
            }
        }
    }

    // initialization of iteration arrays
    for (int x=0;x<numstates;x++) {
        a[0][0][x] = b[0][size-1][x] = a[1][0][x] = (x==0) ? 1.0 : 0.0;
        // extrinsic information from 2-1
    }

    // initialization of extrinsic information array from decoder 2, used in decoder
1
    for (int x=0;x<size;x++)
        L[1][x] = 0.0;

    // 4*Eb/No
    double Lc = (4.0*1.0)/No;

    for (int c=0;c<ITERATIONS;c++) {
        // k from 0 to N-1 instead of 1 to N
        for (int k=0;k<size;k++) {
            // calculate the gamma(s',s);
            for (int input=0;input<2;input++) {
                double uk = (input == 0) ? -1.0 : 1.0;

                for (int s=0;s<numstates;s++) {
                    double xk = (output[input][s] == 0) ? -1.0 : 1.0;

                    gammaE[0][input][k][s]=exp(0.5*Lc*parity1[k]*xk);
                    gamma[0][input][k][s]=exp(0.5*uk*(L[1][k]+Lc*mesg[k]))*gammaE[0]
[input][k][s];
                }
            }
        }

        // calculate the alpha terms
        // from 1 to N-1, 0 is precalculated, N is never used
        for (int k=1;k<size;k++) {
            double temp=0;

            // calculate denominator
            for (int state=0;state<numstates;state++)
                temp += a[0][k-1][fromstate[0][state]]*gamma[0][0][k-1][fromstate[0]
[state]] + a[0][k-1][fromstate[1][state]]*gamma[0][1][k-1][fromstate[1][state]];

            for (int state=0;state<numstates;state++)
/* DIV */       a[0][k][state] = (a[0][k-1][fromstate[0][state]]*gamma[0][0][k-1][fr
omstate[0][state]] + a[0][k-1][fromstate[1][state]]*gamma[0][1][k-1][fromstate[1][sta
te]])/temp;
        }

        // from N-1 to
        for (int k=size-1;k>=1;k--) {
            double temp=0;

            // calculate denominator
            for (int state=0;state<numstates;state++)
                temp += a[0][k][fromstate[0][state]]*gamma[0][0][k][fromstate[0][sta
te]] + a[0][k][fromstate[1][state]]*gamma[0][1][k][fromstate[1][state]];

                for (int state=0;state<numstates;state++)
/* DIV */           b[0][k-1][state] = (b[0][k][tostate[0][state]]*gamma[0][0][k][st
ate] + b[0][k][tostate[1][state]]*gamma[0][1][k][state])/temp;
        }

        for (int k=0;k<size;k++) {
            double min=0;
```


WO 00/07323 PCT/US99/17369

```
            // find the minimum product of alpha, gamma, beta
            for (int u=0;u<2;u++)
            for (int state=0;state<numstates;state++) {
                double temp=a[0][k][state]*gammaE[0][u][k][state]*b[0][k][tostate[u]
[state]];

                if ((temp < min && temp != 0)|| min == 0)
                min = temp;
            }

            // if all else fails, make min real small

            if (min == 0 || min > 1)
                min = 1E-100;

            double topbottom[2];

            for (int u=0;u<2;u++) {
                topbottom[u]=0.0;

                for(int state=0;state<numstates;state++)
                    topbottom[u] += (a[0][k][state]*gammaE[0][u][k][state]*b[0][k][t
ostate[u][state]]);
            }

            if (topbottom[0] == 0)
                topbottom[0] = min;
            else if (topbottom[1] == 0)
                topbottom[1] = min;

            if (topbottom[0] < 1e-100)  topbottom[0] = 1e-100;
            if (topbottom[1] < 1e-200)  topbottom[1] = 1e-200;

if (bHEADER_LETTER) fprintf (stderr, "a");

            double N = topbottom[1];
            double D = topbottom[0];

if (bHEADER_VALUES) fprintf (stderr, "%e / %e : %e", N, D, min);

if (bMINIMUM_NUMERATOR)   if (N < 1e-200) fprintf (stderr, "aN %e", N), N = N;
if (bMINIMUM_DENOMINATOR) if (D < 1e-100) fprintf (stderr, "aD %e", D), D = 1e-100;

            L[0][k] = log(N/D);
        //? L[0][k] = (log(topbottom[1]/topbottom[0]));

if (bHEADER_LETTER) fprintf (stderr, "a");
if (bHEADER_RETURN) fprintf (stderr, "\n");
        }

        interleavedouble(L[0],size);
        // remember to deinterleave for next iteration
        interleavedouble(mesg,size);

        // start decoder 2
        // code almost same as decoder 1, could combine code into one but too lazy

        for (int k=0;k < size; k++) {
            // calculate the gamma(s',s);

            for (int input=0;input<2;input++) {
                double uk = (input == 0) ? -1.0 : 1.0;

                for (int s=0;s<numstates;s++) {
                    double xk = (output[input][s] == 0) ? -1.0 : 1.0;

                    gammaE[1][input][k][s]=exp(0.5*Lc*parity2[k]*xk);
                    gamma[1][input][k][s]=exp(0.5*uk*(L[0][k]+Lc*mesg[k]))*gammaE[1]
[input][k][s];
                }
            }
        }

        // calculate the alpha terms
        for (int k=1;k<size;k++) {
            double temp=0;

            // calculate denominator
            for (int state=0; state < numstates; state++)
                temp += a[1][k-1][fromstate[0][state]]*gamma[1][0][k-1][fromstate[0]
[state]] + a[1][k-1][fromstate[1][state]]*gamma[1][1][k-1][fromstate[1][state]];

            for (int state=0;state<numstates;state++)
/* DIV */       a[1][k][state] = (a[1][k-1][fromstate[0][state]]*gamma[1][0][k-1][fr
omstate[0][state]] + a[1][k-1][fromstate[1][state]]*gamma[1][1][k-1][fromstate[1][sta
te]])/temp;
        }

        // in the first iteration, set b[1][N-1] = a[1][N-1] for decoder 2
        // this decoder can't be terminated to state 0 because of the interleaver
        // the performance loss is supposedly negligible.
        if (c==0) {
            double temp=0;

            // calculate denominator
            for (int state=0;state<numstates;state++)
                temp += a[1][size-1][fromstate[0][state]]*gamma[1][0][size-1][fromst
ate[0][state]] + a[1][size-1][fromstate[1][state]]*gamma[1][1][size-1][fromstate[1][s
tate]];

            for (int state=0;state<numstates;state++)
/* DIV */       b[1][size-1][state] = (a[1][size-1][fromstate[0][state]]*gamma[1][0]
[size-1][fromstate[0][state]] + a[1][size-1][fromstate[1][state]]*gamma[1][1][size-1]
[fromstate[1][state]])/temp;
        }

        for (int k=size-1;k>=1;k--) {
            double temp=0;

        // calculate denominator
        for (int state=0;state<numstates;state++)
            temp += a[1][k][fromstate[0][state]]*gamma[1][0][k][fromstate[0][state]]
 + a[1][k][fromstate[1][state]]*gamma[1][1][k][fromstate[1][state]];

        for (int state=0;state<numstates;state++)
/* DIV */   b[1][k-1][state] = (b[1][k][tostate[0][state]]*gamma[1][0][k][state] + b
[1][k][tostate[1][state]]*gamma[1][1][k][state])/temp;
        }

        for (int k=0;k<size;k++) {
            double min = 0;
```


WO 00/07323 PCT/US99/17369

```
            // find the minimum product of alpha, gamma, beta

            for (int u=0;u<2;u++) {
                for (int state=0;state<numstates;state++) {
                    double temp=a[1][k][state]*gammaE[1][u][k][state]*b[1][k][tostat
e[u][state]];

                    if ((temp < min && temp != 0)|| min == 0)
                        min = temp;
                }
            }

            // if all else fails, make min real small

            if (min == 0 || min > 1)
                min = 1E-100;

            double topbottom[2];

            for (int u=0;u<2;u++) {
                topbottom[u]=0.0;

                for (int state=0;state<numstates;state++)
                    topbottom[u] += (a[1][k][state]*gammaE[1][u][k][state]*b[1][k][t
ostate[u][state]]);
            }

            if (topbottom[0] == 0)
                topbottom[0] = min;
            else if (topbottom[1] == 0)
                topbottom[1] = min;

            if (topbottom[0] < 1e-100)  topbottom[0] = 1e-100;
            if (topbottom[1] < 1e-200)  topbottom[1] = 1e-200;

if (bHEADER_LETTER) fprintf (stderr, "b");

            double N = topbottom[1];
            double D = topbottom[0];

if (bHEADER_VALUES) fprintf (stderr, "%e / %e : %e", N, D, min);

if (bMINIMUM_NUMERATOR)   if (N < 1e-200) fprintf (stderr, "aN %e", N), N = N;
if (bMINIMUM_DENOMINATOR) if (D < 1e-100) fprintf (stderr, "aD %e", D), D = 1e-100;

            L[1][k] = log(N/D);
        //? L[1][k] = (log(topbottom[1]/topbottom[0]));

if (bHEADER_LETTER) fprintf (stderr, "b");
if (bHEADER_RETURN) fprintf (stderr, "\n");
        }

        deinterleavedouble(mesg,size);
        deinterleavedouble(L[1],size);

        // get L[0] back to normal after decoder 2

        deinterleavedouble(L[0],size);

        bool temp=true;
        for (int k=0;k<size;k++)
            if (boolmesg[k] != ((Lc*mesg[k] + L[0][k] + L[1][k] > 0.0) ? true : fals
e))
                temp = false;

        // we can quit prematurely since it has been decoded

        if (temp==true) {
            returnvalue = c;
            c=ITERATIONS;
        }
    }
    // end decoder 2
    // make decisions

    for (int k=0;k<size;k++)
        if ((Lc*mesg[k] + L[0][k] + L[1][k]) > 0)
            mesg[k] = 1.0;
        else
            mesg[k] = -1.0;

    return returnvalue;
}

void deinterleavedouble(double *mesg, unsigned size)
{
    double *temp;

    temp = new double[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[deinterleavearray[x]] = temp[x];

    delete temp;
}

void interleavedouble(double *mesg, unsigned size)
{
    double *temp;

    temp = new double[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[interleavearray[x]] = temp[x];

    delete temp;
}

void interleave(bool *mesg,unsigned size)
{
    bool *temp;

    temp = new bool[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];
```


WO 00/07323 PCT/US99/17369

```
    for (int x=0;x<size;x++)
        mesg[interleavearray[x]] = temp[x];

    delete temp;
}

void deinterleave(bool *mesg,unsigned size)
{
    bool *temp;

    temp = new bool[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[deinterleavearray[x]] = temp[x];

    delete temp;
}

void createinterleave(unsigned size)
{
    bool *yesno;

    yesno = new bool[size];

    for (int x=0;x<N;x++)
        yesno[x]=false;

    // create an interleave array
    for (int x=0;x<N;x++) {
        unsigned  val;

        do {
            val=before.longrandom(N);
        } while(yesno[val] == true);

        yesno[val] = true;
        interleavearray[x] = val;
        deinterleavearray[val] = x;
    }

    delete yesno;
}

void encode(bool *mesg,bool *parity,unsigned size, bool force)
{

    unsigned state=0;

    for (int x=0;x<size;x++) {

        // force the encoder to zero state at the end

        if (x>=size-memory && force) {
            if (tostate[0][state]&1)
            mesg[x] = true;
        else
            mesg[x] = false;
        }

        // can't assume the bool type has an intrinsic value of 0 or 1
        // may differ from platform to platform

        int uk = mesg[x] ? 1 : 0;

        // calculate output due to new mesg bit

        parity[x] = output[uk][state];

        // calculate the new state

        state = tostate[uk][state];
    }
}

bool add(bool a, bool b)
{
    return a==b ? false : true;
}

double gaussian(double variance)
{
    // static because we don't want to have it initialized each time we go in

    double returnvalue=0;
    double k;

    k = sqrt(variance/2.0);

    // add 24 uniform RV to obtain a simulation of normality

    for (int x=0;x<24;x++)
        returnvalue += before.doublerandom();

    return k*(returnvalue-0.5*24);
}

////////////////////////////////////////////////////////////////////////////
//
//  fjh this file was built from tm.cpp
//

extern int tx_bits;
extern int dl_bits;
extern int rx_bits;

extern double tx_A;
extern double tx_B;
extern double tx_C;

extern double ln_A;
extern double ln_B;
extern double ln_C;

extern FILE *spcout;

bool *tx_message;
bool *tx_parity1;
bool *tx_parity2;
```

WO 00/07323 PCT/US99/17369

```
bool *rx_message;
bool *dl_message;

double *tx_message_A;
double *tx_parity1_B;
double *tx_parity2_C;

double *no_message_A;
double *no_parity1_B;
double *no_parity2_C;

double *rx_message_A;
double *rx_parity1_B;
double *rx_parity2_C;

void
alloc_turbo_memory (unsigned length, unsigned iterations)
{
    fprintf (stderr, "allocated memory %d\n", length);

    N = length;
    ITERATIONS = iterations;

    interleavearray   = new unsigned[N];
    deinterleavearray = new unsigned[N];

    tx_message = new bool [N];
    tx_parity1 = new bool [N];
    tx_parity2 = new bool [N];

    rx_message = new bool [N];
    dl_message = new bool [N];

    tx_message_A = new double [N];
    tx_parity1_B = new double [N];
    tx_parity2_C = new double [N];

    no_message_A = new double [N];
    no_parity1_B = new double [N];
    no_parity2_C = new double [N];

    rx_message_A = new double [N];
    rx_parity1_B = new double [N];
    rx_parity2_C = new double [N];

    createencodetable();
}

void
free_turbo_memory (void)
{
    fprintf (stderr, "freed memory\n");

    delete interleavearray;
    delete deinterleavearray;

    delete tx_message;
    delete tx_parity1;
    delete tx_parity2;

    delete rx_message;
    delete dl_message;

    delete tx_message_A;
    delete tx_parity1_B;
    delete tx_parity2_C;

    delete no_message_A;
    delete no_parity1_B;
    delete no_parity2_C;

    delete rx_message_A;
    delete rx_parity1_B;
    delete rx_parity2_C;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

int  INC = 0;

unsigned numloops    = 0;
unsigned totaliter   = 0;
unsigned totalN      = 0;
unsigned totalerrors = 0;

void
init_turbo (double sigma)
{
    fprintf (stderr, "initialized memory\n");

    No = sigma*sigma;

    INC = 0;

    numloops    = 0;
    totaliter   = 0;
    totalN      = 0;
    totalerrors = 0;

    for (int x=0;x<N;x++) {
        tx_message [x] = 0;
        tx_parity1 [x] = 0;
        tx_parity2 [x] = 0;

        rx_message [x] = 0;
        dl_message [x] = 0;

        tx_message_A [x] = 0;
        tx_parity1_B [x] = 0;
        tx_parity2_C [x] = 0;

        no_message_A [x] = 0;
        no_parity1_B [x] = 0;
        no_parity2_C [x] = 0;

        rx_message_A [x] = 0;
        rx_parity1_B [x] = 0;
        rx_parity2_C [x] = 0;
    }
}
```

21.

WO 00/07323 PCT/US99/17369

```
void
turbo_encode (void)
{
    tx_message [INC] = (tx_bits == 1);

    tx_A = 0.0;
    tx_B = 0.0;
    tx_C = 0.0;
}

void
turbo_decode (int blind_decode)
{
    no_message_A [INC] = ln_A;
    no_parity1_B [INC] = ln_B;
    no_parity2_C [INC] = ln_C;

    ////////////////////////////////////////////////////////////////////////

    tx_A = tx_message_A [INC];
    tx_A = tx_parity1_B [INC];
    tx_A = tx_parity2_C [INC];

    ln_A = rx_message_A [INC];
    ln_B = rx_parity1_B [INC];
    ln_C = rx_parity2_C [INC];

/// tx_bits = tx_message [INC] ? 1 : 0;
    dl_bits = dl_message [INC] ? 1 : 0;
    rx_bits = rx_message [INC] ? 1 : 0;

    if (++INC < N-4) return;

    ////////////////////////////////////////////////////////////////////////

    INC = 0;

    createinterleave(N);

    encode(tx_message,tx_parity1,N,true);
    interleave(tx_message,N);
    encode(tx_message,tx_parity2,N,false);
    deinterleave(tx_message,N);

    for (int x=0;x<N;x++) {
        tx_message_A [x] = tx_message [x] ? +1.0 : -1.0;
        tx_parity1_B [x] = tx_parity1 [x] ? +1.0 : -1.0;
        tx_parity2_C [x] = tx_parity2 [x] ? +1.0 : -1.0;

        rx_message_A [x] = tx_message_A [x] + no_message_A [x];
        rx_parity1_B [x] = tx_parity1_B [x] + no_parity1_B [x];
        rx_parity2_C [x] = tx_parity2_C [x] + no_parity2_C [x];

        dl_message [x] = (tx_message_A [x] >= 0.0);
        rx_message [x] = (rx_message_A [x] >= 0.0);
    }

    if (blind_decode == 1) return;

    ////////////////////////////////////////////////////////////////////////

    unsigned numiter;
    unsigned numerrors=0;

    numiter = decode (rx_message_A, rx_parity1_B, rx_parity2_C, N, dl_message);

    for (int x=0;x<N;x++)
        rx_message [x] = (rx_message_A [x] >= 0.0);

    for (int x=0;x<N;x++)
        if (dl_message [x] != rx_message [x]) numerrors++;

    totalerrors += numerrors;
    totalN      += N;
    totaliter   += numiter;
    numloops++;

    fprintf (spcout, "%6d - Terr %6d Tcnt %6d Aitr %5.1f - err %3d cnt %3d itr %5.1f
\n"
        , numloops
        , totalerrors
        , totalN
        , (double) totaliter / numloops
        , numerrors
        , N
        , (double) numiter
    );
    fprintf (stderr, "%6d - Terr %6d Tcnt %6d Aitr %5.1f - err %3d cnt %3d itr %5.1f
\n"
        , numloops
        , totalerrors
        , totalN
        , (double) totaliter / numloops
        , numerrors
        , N
        , (double) numiter
    );
}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
```


WO 00/07323 PCT/US99/17369